conference

proceedings

2nd USENIX Symposium on Internet Technologies and Systems

Boulder, Colorado, USA
October 11–14, 1999

Sponsored by
USENIX
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION
Co-Sponsored by
IEEE Computer Society
Technical Committee on the Internet

For additional copies of these proceedings contact:

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 USA
Phone: 510 528 8649
FAX: 510 548 5738
Email: office@usenix.org
WWW URL: http://www.usenix.org

The price is $24 for members and $30 for nonmembers.
Outside the U.S.A. and Canada, please add
$12 per copy for postage (via air printed matter).

Past USITS Proceedings

1st USITS Symposium	1997	Monterey, California	$20/26

ISBN 1-880446-26-X

Printed in the United States of America on 50% recycled paper, 10-15% post consumer waste.

USENIX Association

Proceedings of the 2nd USENIX Symposium on Internet Technologies and Systems (USITS '99)

October 11–14, 1999
Boulder, Colorado, USA

Symposium Organizers

Program Chair

Fred Douglis, *AT&T Labs—Research*

Program Committee

Eric A. Brewer, *University of California at Berkeley and Inktomi*
Peter Honeyman, *CITI, University of Michigan*
David B. Johnson, *Carnegie Mellon University*
P. Krishnan, *Bell Labs, Lucent Technologies*
Geoffrey H. Kuenning, *Harvey Mudd College*
Yoelle Maarek, *IBM Haifa Research Lab*
Udi Manber, *Yahoo! Inc.*
Jeffrey Mogul, *Compaq Western Research Laboratory*
Katia Obraczka, *University of Southern California Information Sciences Institute*

External Reviewers

Mark Crovella
Sukumal Imudom
Terence Kelly
Balachander Krishnamurthy
Qi Lu
David Maltz
Danny Raz
Fabio Silva
Cormac Sreenan
Carl Staelin
Dan Suciu
Win Treese
Walter Willinger

The USENIX Association Staff

CONTENTS

2nd USENIX Symposium on Internet Technologies and Systems

October 11–14, 1999
Boulder, Colorado, USA

Wednesday, October 13

Proxy Implementation
Session Chair: Jeffrey Mogul, Compaq Western Research Laboratory

Prefetching
Session Chair: Geoffrey H. Kuenning, Harvey Mudd College

Architectures
Session Chair: David B. Johnson, Carnegie Mellon University

Thursday, October 14

Caching Policies
Session Chair: Katia Obraczka, University of Southern California Information Sciences Institute

Server Implementation
Session Chair: Fred Douglis, AT&T Labs—Research

Index of Authors

Message from the Symposium Chair

Welcome to the 1999 USENIX Symposium on Internet Technologies and Systems. Having moved from helping to kick off the 1997 USITS to chairing the 1999 conference, I am pleased to see how USITS has quickly matured into a premier forum for the discussion of important issues surrounding the Internet, its underlying infrastructure, and the Internet applications that now proliferate.

The program committee selected 22 of 67 submissions. As with the previous USITS, Web caching and other uses of Web proxies are a common theme in the conference. About half the papers fall into this category. The remaining papers cover a broad spectrum of interesting topics, including architectural and implementation issues and higher-level applications. Udi Manber, who also served on the program committee, will open the conference with a keynote address using his perspective as Chief Scientist of Yahoo! Inc. to discuss the future of e-commerce.

I would like to thank those who provided their assistance to me and to the conference. There are no doubt others, not included here, who also deserve our thanks; I apologize in advance for any omissions.

For the second year in a row as an organizer of a USENIX conference, I have seen the USENIX staff demonstrate its collective wisdom and efficiency by working behind the scenes to make the conference a success. I would like to thank Ellie Young, Judy DesHarnais, Moun Chau, Jane-Ellen Long, Monica Ortiz, Jennifer Radtke, and Toni Veglia. Dan Klein did his usual excellent job of organizing conference tutorials. Peter Honeyman served as USENIX board liaison and as a member of the program committee, providing valuable guidance on the organization of the conference, selection of the program committee, and other matters.

I thank my employer, AT&T Labs, and my management, Michael Merritt and Hamid Ahmadi, for supporting my work with the conference. The company's support in matters such as these, not just for me but for several members of the USENIX community, is very much appreciated.

Thanks to the program committee and other reviewers for their work in reviewing a large number of papers in a short time. I am particularly grateful to several people who contributed additional reviews at the last minute when another reviewer was unable to provide them. I also very much appreciate the flexibility of the program committee, as well as those who submitted papers, in handling the uncertainties and schedule shuffling that resulted from the unexpected early birth of my son Pierce.

This leads me finally to the most important thank-you of all: to my wife, Lisa, for putting up with all the USITS obligations in the heat of battle. She entered the hospital with early labor just after the submissions were received, and delivered our son in the heat of the reviewing process—days before the program committee meeting. Our timing might have been better, but we're grateful everything turned out well.

Fred Douglis, Program Chair

September 1999

Scalable Web Caching of Frequently Updated Objects using Reliable Multicast

Dan Li and David R. Cheriton
Stanford University

Abstract

Frequently updated web objects reduce the benefit of caching, increase the problem of cache inconsistency, and aggravate the inefficiency of the conventional "repeated unicast" delivery model. In this paper, we investigate multicast invalidation and delivery of popular, frequently updated objects to web cache proxies. Our protocol, MMO, groups objects into volumes, each of which maps to one IP multicast group. We show that, by forming volumes of the appropriate size and/or object correlation, the benefit from reliable multicast outweighs the cost of delivering extraneous data as well as the overhead of multicast reliability. Moreover, trace-driven simulations show that the bandwidth saving over conventional approaches increases significantly as the audience size grows. We conclude that MMO provides efficient bandwidth utilization and service scalability, and makes strong web cache consistency for dynamic objects practical.

1. Introduction

Web proxy caching [4] is critical to the continuing success of the Web. It improves the response time and reduces the load on the network and web servers. The falling cost of memory and disk allows web cache proxies to hold an increasing amount of web content. As the Web carries more web objects[1] that are *both* accessed and modified frequently, the hit rate of web caches is limited more by consistency than by cache capacity. Cached copies of frequently updated objects become *stale* more often. Frequently retrieving new copies defeats the benefit of caching.

Frequently updated objects also raise the consistency protocol overhead. With web cache consistency protocols such as adaptive TTL (Time-To-Live) [15], the rate of polling by the proxy must be considerably higher than the rate of modification at the web server in order to maintain an acceptable *stale rate* (percentage of instances that the cache returns a stale document). For frequently updated objects such as sports and financial news that change several times a day, polling overhead can be excessive for the network and the web server. Alternatively, the web server can send cache invalidations to web caches. Cao et. al. [7] performed an excellent study on a TCP-based invalidation protocol, concluding that strong cache consistency can be maintained with little or no extra cost over the current weak-consistency approaches. However, the web server has to keep per-proxy state and establish TCP connections to all of the proxies to deliver the invalidation, a significant overhead for widely cached objects. Moreover, after the invalidation, there is likely to be a sudden influx of requests from many caches (triggered by client requests or prefetching [24]), potentially saturating the server and causing link congestion. These bursts of requests may produce peak loads comparable to that experienced without caching. If servers are engineered for these peak loads, the benefit of caching for servers is minimal. Fundamentally, the "repeated unicast" delivery model does not scale.

Addressing these problems, we propose MMO — multicast invalidation followed by multicast delivery of a *volume* of web objects to web cache proxies using the OTERS reliable multicast protocol [20]. The cost of OTERS in this context is evaluated using NS [30]. We study MMO's performance using trace-driven simulations. From these studies, we conclude that MMO is far more scalable than conventional and hybrid protocols and provides strong cache consistency, fast responses, and efficient bandwidth utilization.

The rest of the paper is organized as follows. Section 2 describes our proposed protocol. Section 3 discusses a number of alternatives to be compared with our approach. Section 4 outlines the simulation environment. Section 5 assesses the cost of unicast and multicast transport protocols. Section 6 analyzes the performance of our protocol. Section 7 discusses related work. Section 8 concludes the paper. Appendix A describes web access traces. Appendix B describes the process of measuring the transport protocols.

2. A Multicast-based Web Caching Protocol

In MMO, the web server multicasts cache invalidations

1 A web object consists of one or more files a browser needs to retrieve from the web server in order to display a URL. A web server (or server) refers to a web content source; a web cache proxy (also as a cache or a proxy) refers to a shared URL cache for a group of local web clients, e.g., hosts within an ISP network or a corporate LAN.

and modified objects to a multicast group using the OTERS reliable multicast protocol. Web caches subscribe to the multicast group to receive the information.

2.1. OTERS

OTERS (On-Tree Efficient Recovery using Subcast) [20] organizes group members into hierarchical subgroups by exchanging session messages to elect a *designated receiver* or *DR* for each subgroup. DRs then employ *subcasting*[2] for local retransmission. Figure 2.1 shows its recovery process.

The notification mode (OTERS-NT) uses ACKs to reliably deliver notifications such as web cache invalidations. Upon receiving a notification, the group member sends an ACK to its DR, which in turn sends an ACK to its own DR. Each DR retransmits the notification to non-responding subgroup members.

The file transfer mode (OTERS-FT) is designed for files. A receiver learns about parameters of the file transfer from a prior notification (e.g., a web cache invalidation), including the starting time, the file length and the transmission rate. At the end of the file transmission, if the receiver has missed any packets, it sends a NAK (containing the sequence numbers of all missing packets) to its DR for retransmissions.

2.2. The Invalidation Phase

In any invalidation protocol, the web server sends invalidation messages to web caches when an object is modified. Each web cache then deletes the cached copy (if one is cached). In the presence of network or process failures, *leases* or *volume leases* [12, 33] can serve as an efficient fault-tolerance mechanism. Currently the HTTP protocol does not support invalidation but the server part can be implemented in the HTTP accelerator [8] (a form of web caching at the server location). This way, invalidation becomes part of a signaling protocol between web caches such as ICP [32].

In MMO, an *invalidation channel* is a multicast address that web cache proxies subscribe to. The web server uses OTERS-NT to notify the channel subscribers of modifications to a volume of objects. A proxy that is not subscribed to the invalidation channel still requests those objects directly from the web server, which indicates in the response that an invalidation channel exists.

[2] Subcasting is multicasting of a packet over a subtree of the multicast delivery tree. One subcast retransmission can repair an entire subtree's losses that are caused by one packet drop at the root of the subtree. OTERS is built on IP encapsulation [26] and IGMP traceroute [10], with security extensions that involve router changes but impose no additional state and little processing overhead.

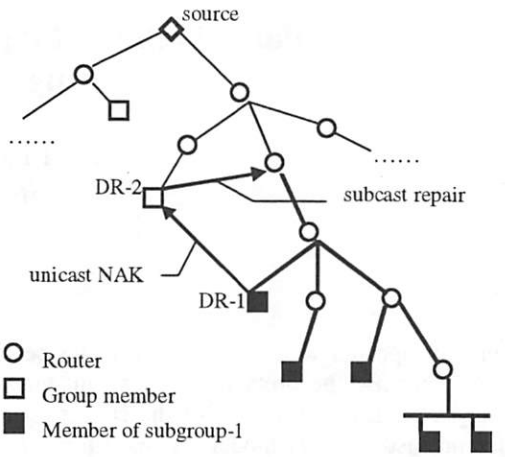


Figure 2.1 OTERS. *Designated Receiver* DR-1 unicasts a NAK (negative acknowledgment) to its own designated receiver – DR-2 – after detecting a packet loss. DR-2 responds with a repair to DR-1's subtree, assuming DR-2 received this packet. Bold links indicate the path of the retransmission.

The proxy can join the channel when enough number of objects in the volume is cached.

The invalidation channel is expected to be long-lived and have relatively stable memberships. For example, a proxy may stay in the channel for 12 hours or longer. Using OTERS also means that subscribed proxies are committed to exchanging information and maintaining the subgroup hierarchy. Fortunately, web caches are generally stable, well maintained, and well connected to the Internet. The opposite is dial-up or wireless users, for whom our scheme is not suitable.

2.3. The Delivery Phase

After a multicast invalidation, the server multicasts the modified object via OTERS-FT to the invalidation channel, which subscribed proxies receive and then return to clients in subsequent requests.

A *volume* is a set of objects that share the same invalidation channel. Having multiple objects per volume is more efficient than having one object per volume because the *multicast overhead* (including address allocation, routing and transport session organization) can be amortized over more objects.

A multi-object volume introduces extraneous data. For instance, one may receive from the multicast channel a message that invalidates an object it does not cache, or receive an object that is modified again before any client requests the object. The server, however, can reduce the amount of extraneous data by limiting the volume size and assigning related objects to the same volume (so that a proxy is likely to cache most of them). The server can form volumes based on access statistics,

Acronym	*Invalidation Method*	*Delivery Method*
MMO	Multicast via OTERS-NT	proactive Multicast via OTERS-FT
MMF	Multicast via OTERS-NT	proactive Multicast via Digital Fountain
UMF	Unicast via TCP	proactive Multicast via Digital Fountain
MU	Multicast via OTERS-NT	on-demand Unicast via TCP
UU	Unicast via TCP	on-demand Unicast via TCP
AT	Adaptive TTL	on-demand Unicast via TCP
PET	Polling-Every-Time	on-demand Unicast via TCP

Table 1 Acronyms of the seven web caching protocols

URL prefixes, content subjects, etc. [9]. Furthermore, proxies are less prone to extraneous data than end-users because a proxy aggregates requests from many end-users, raising the traffic and hit rate of popular objects.

2.4. The Pros and Cons of MMO

MMO offers several significant advantages. First, multicast invalidation provides strong cache consistency[3] without any per-proxy state at the server and without aggressive polling by the proxy. Second, proactive multicast updates provide lower web access time than on-demand unicast delivery. Third, multicast invalidation and delivery are more scalable to large audiences than their unicast counterparts.

However, proactive multicast is not always more efficient than on-demand unicast because of multicast overheads and extraneous data. But MMO compensates for these potential drawbacks by employing one channel for both cache invalidation and object delivery to amortize multicast overheads. MMO also relies on efficient volumes to control the amount of extraneous data. Hence MMO is more efficient when delivering popular, frequently modified and correlated web objects in a volume to a large number of web caches. For example, the CNNfn.com homepage and top stories can be disseminated in a volume using MMO.

3. Other Web Caching Protocols

To set the stage for comparing MMO with other alternatives, we first introduce some hybrid protocols (MMF, MU, UMF and UU) along with the traditional ones (AT and PET). Table 1 lists their main features.

3.1. Hybrid Web Caching Protocols

MMF and MU use multicast invalidation, similar to MMO. Conversely, UMF and UU use *unicast invalidation* [7]. The web server keeps a list of web caches that have requested an object since its last modification. When the object is modified, the server sends an invalidation (via TCP) to each cache on the list.

MU and UU use *on-demand TCP delivery*. After the invalidation (either unicast or multicast), the proxy deletes the invalidated copy and retrieves a fresh copy only when the next client request arrives. There is no extraneous data but the server has to repeatedly unicast the object on demand.

MMF and UMF use *proactive multicast delivery via Digital Fountain*[4]. After the invalidation (either unicast or multicast), the server multicasts the modified object via Digital Fountain to a *delivery channel*, a multicast group that is allocated for delivering this object). The delivery channel is short-lived. Proxies can decide whether to join it. If a proxy joins the channel, it receives a copy and returns this copy to clients upon future requests. Otherwise, it retrieves a fresh copy (via TCP) when the next client request arrives.

The delivery channel allows a tradeoff between unicast and multicast delivery methods, in the amount of extraneous data a proxy chooses to receive. The prefetch decision is based on the probability of a future client request coming before the next modification. We use the following policy. Define a *join threshold W*. If no client has requested the object for the time spanning the last W invalidations, the proxy does not join the delivery channel. Otherwise, it does. With this flexibility, MMF incurs the multicast overhead on each delivery. Section 6 shows that MMO in fact outperforms MMF.

To efficiently support the above schemes, there are two requirements on multicast routing. First, the invalidation channel is long-lived and requires efficient routing state maintenance, e.g., limiting membership heartbeats to occur only at the leaves. Second, the delivery channel is short-lived and requires fast join/leave and scal-

[3] There is a small window of opportunity (from the creation of cache invalidation to the completion of object delivery) for clients to get the slightly obsolete copy from the proxy.

[4] Digital Fountain [5, 29] is designed for bulk data transfer. The source encodes an entire file using Forward Error Correction codes and multicasts it continuously by looping through the encoded data. A receiver tunes to the multicast channel at any time and leaves the channel as soon as it receives enough encoded packets in order to reconstruct the original file. The source can stop sending once the multicast group is empty or after having looped several times.

able address allocation. These requirements are consistent with the research community's effort on multicast routing and are met by proposals such as EXPRESS single-source multicast routing [17].

3.2. Traditional Web Caching Protocols

Polling-every-time provides strong cache consistency, like all the above protocols. The proxy always sends an "If-Modified-Since" request to the server before returning any cached copy to clients. The server responds with either a modified copy or "Not Modified". The latter case is called a *slow hit* because the cached copy is returned to the client after a round trip to the server. Conversely, in an invalidation-based protocol, all hits are *fast hits* because the cached copy is immediately returned to the client.

Adaptive TTL [15] provides weak cache consistency and is based on the observation that "older" files are less likely to be modified. The proxy sets the TTL of a cached copy to α times the "age" of the object (i.e., from its last modification to now). By default, α is 0.2 in Squid [32] and 0.5 in Harvest [4]. Before the TTL expires, client requests are served directly from the cache. They are fast hits but may be stale. Upon the first client request after the TTL expires, the proxy sends an "If-Modified-Since" request to the server. The result may be a modified copy or a slow hit. Then the TTL of the cached copy is adjusted accordingly.

4. The Simulation Environment

4.1. Web Access Traces

The simulation uses three types of traces. One is the *Surge trace*, generated by the Surge HTTP request generator [2]. Second is the *Stanford trace*, the server log of Stanford University's official web site. Third is the *NLANR trace*, proxy logs of accesses to *CNN.com* by the 8 top-domain proxies in the NLANR (National lab of Applied Network Research) Cache Hierarchy [23]. *Appendix A* describes these traces in more detail.

Generate modifications. The traces do not provide the object modification history so we adopt the *hot/cold* model [7] to generate modifications. First, 1% of the web objects are picked uniformly across the object popularity ranks as the frequently updated (or hot) objects. Then, given k hot objects and an average object lifetime of L seconds, every L / k seconds the modification generator randomly picks one from the k objects to modify. This leads to a geometric lifetime distribution.

Volume formation. The Surge and Stanford traces use *random formation*. In other words, a volume of size V consists of the V most popular, frequently updated objects. The NLANR trace uses *prefix formation*. The volume consists of six objects that share the URL prefix "*http://www.CNN.com/WORLD/meast/9812/17/iraq.strike*".

4.2. Join Decision and Caching Decision

A proxy joins the invalidation channel if it caches at least one object in the volume. In reality, a proxy may decide to join the channel only after a few objects in the volume are cached. Our assumption is more conservative in that it results in more extraneous data.

After the proxy joins the channel, any object in the volume is cached once accessed. This decision is realistic because objects in the volume are popular (based on the server's statistics) and warrant caching. Caching objects in the volume that are less popular to a proxy presents only disk space cost and no extra consistency cost. With cheap disks and RAM, a deep cache or cache farm can afford the space in exchange for lower bandwidth consumption and better response time to the end-users. A volume-wise caching decision also does not reduce the hit rate in a deep cache because a cache often can reach a size beyond which the hit rate does not rise much by adding more cache space. For example, a 24GB cache is sufficient for a daily web flow of 100 gigabits (according to the ISP-caching mailing list).

4.3. Performance Metrics

The web-caching simulation uses the following performance metrics:

response time: the time from when the proxy receives a client request to the time it finishes responding. This metric reflects the user-perceived web access time because the way a client contacts its proxy is the same regardless of the web caching protocol.

stale rate: the percentage of responses a proxy returns to its clients that contain stale data. Only adaptive TTL has a non-zero stale rate. All other protocols offer strong consistency and therefore zero stale rates.

packet count: the number of distinct packets exchanged among the web server and proxies in order to fulfill the client requests. Packets that a proxy sends to its clients are not counted because all the protocols incur the same cost. The packet size is assumed to be 1024 bytes, a compromise between two popular network packet sizes: 550 and 1500 bytes.

packet-hop count: the number of hops the packets tra-

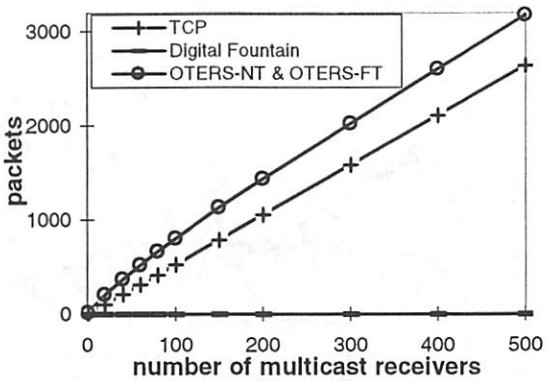


Figure 5.1 Session Overhead in packets

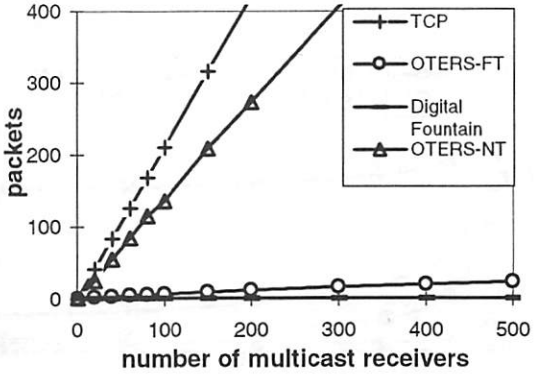


Figure 5.2 Per-Packet Cost in packets

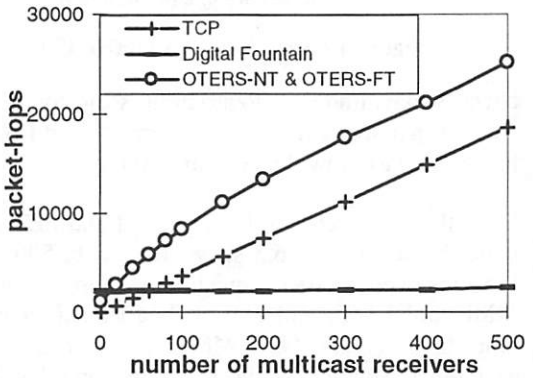


Figure 5.3 Session Overhead in packet-hops

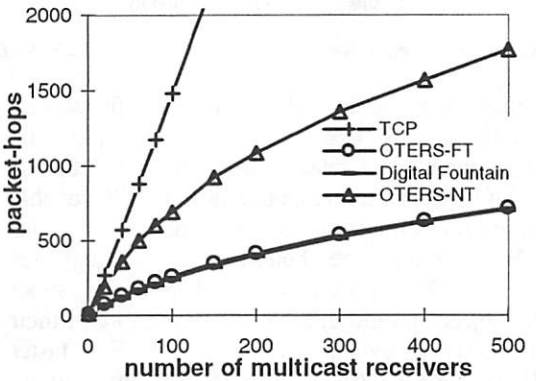


Figure 5.4 Per-Packet Cost in packet-hops

verse between the server and proxies, reflecting the amount of wide-area traffic a caching protocol imposes.

5. Performance of the Transport Protocols

To assess the traffic load of the various web-caching protocols, we developed a traffic load model for the transport protocols TCP, OTERS and Digital Fountain. *Appendix B* describes the measurement process on the simulator NS [30]. Measurements show that the traffic load of a transport session can be modeled as $Load(m,n) = f(m) + n \bullet g(m)$, where $f(m)$ is the session overhead, $g(m)$ is the per-packet cost, m is the number of multicast receivers and n is the number of payload packets.

TCP's overhead and per-packet cost are linear in m. TCP's overhead comes from the 3-way handshake and the connection termination. The overhead of OTERS comes from organizing the subgroup hierarchy. The overhead of Digital Fountain comes from packets that the network delivers after a receiver has received all that are necessary to reconstruct the original file but before its leave message is propagated all the way up the multicast delivery tree. The higher rate the source transmits, or the slower the leave message propagates, the more Digital Fountain overhead. Additional overhead may come from flooding of the initial multicast packet, e.g., in a DVMRP routing domain [31].

Figures 5.1 and 5.2 plot the packet counts of the session overhead and the per-packet cost respectively. Figures 5.3 and 5.4 plot the packet-hop counts. The session overhead of Digital Fountain is significantly less than that of TCP and OTERS. However, in MMF, the Digital Fountain overhead is amortized over a single delivery, while in MMO the OTERS overhead is amortized across multiple deliveries. The per-packet costs of OTERS-FT and Digital fountain are similar and much lower than that of TCP and OTERS-NT because the former two use NAKs while the latter two use ACKs.

6. Web Caching Performance Analysis

The traces were replayed through a web caching simulator that implements the 7 protocols (see also Table 1). Performance data is gathered over requests to objects in a volume. Requests outside the volume were not considered. Every set of results has three parameters: V — the number of objects in the volume, P — the number of proxies, and L — the average object lifetime (in minutes). In adaptive TTL, α is set to 0.25. In MMF and UMF, the join threshold W is set to 1.

6.1. From the Client's Perspective

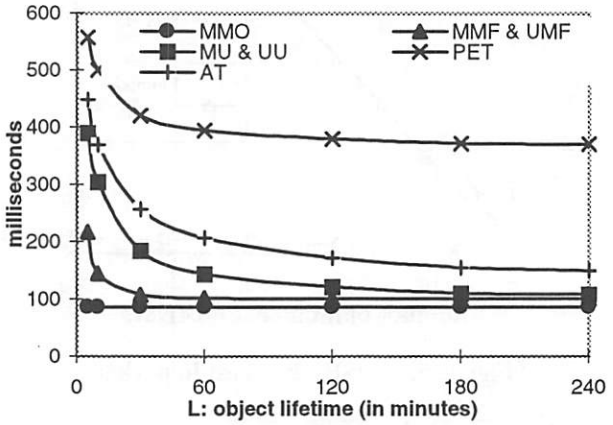


Figure 6.1 Average Response Time (NLANR, P=8, V=6)

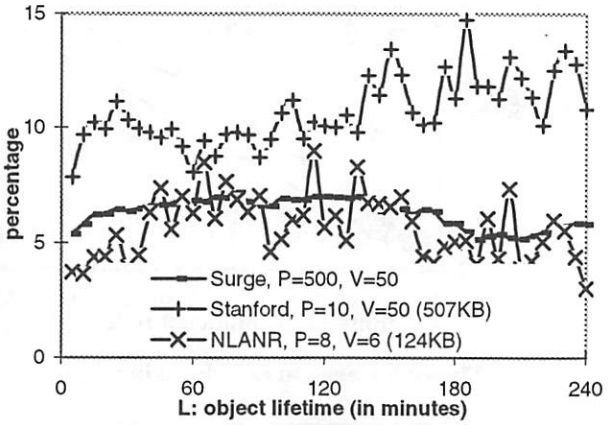


Figure 6.2 Stale Rate of Adaptive TTL

The response time and the stale rate quantify the service quality that end-users experience. Figure 6.1 plots the average response time relative to L, for the NLANR trace. MMO reduces the response time to 57% of that of AT when L = 4 hours and to 34% when L = 30 minutes. MMO sets the lower bound because it generates only fast hits. Other protocols' curves, however, shoot up as the object lifetime shortens, causing more of their fast hits become slow hits or misses. MMF is faster than MU because it retrieves objects sometimes proactively and sometimes on demand. AT polls the server once TTL expires and may discover the document is not modified. Therefore AT has a higher response time than all the invalidation-based protocols. PET is the slowest because it polls the server on every request.

Figure 6.2 plots the stale rate of AT for the three traces. It shows that, with α = 0.25, AT can reach a stale rate of 5% to 15% for objects modified more than once every four hours. The Stanford trace has a higher stale rate because it directly records the end-users' access pattern and hence has more *clustered requests* (requests to the same object, e.g., a course's announcement page, that occur within a short interval, e.g., 3 hours). With clustering, more requests occur before the TTL of the cached copy expires and are subject to stale responses. Conversely, requests to NLANR top-domain caches are filtered by lower-level caches. Requests generated by Surge are also relatively spaced out.

6.2. From the Server's Perspective

Packet counts indicate the amount of traffic that servers and caches have to generate to deliver the web content. Figures 6.3 and 6.4 plot the packet count vs. L and V respectively for the Surge trace. Figures 6.5 and 6.6 plot the same for the Stanford trace. Figures 6.7 and 6.8 plot the packet count vs. P for Surge and NLANR respectively. The figures' Y axes vary in their ranges but all cover 3 magnitudes in logarithmic scale for easy relative comparison, except that Figures 6.7 and 6.8 use a linear scale to show the tangent of the curves.

Overall, MMO sets the lower bound and PET sets the upper bound. Figure 6.3 shows that, with 500 proxies, MMO is over an order of magnitude more efficient than MMF and UMF, and almost 2 orders of magnitudes more than AT and PET. MMF and UMF are close to each other. So are MU and UU, indicating that delivery (as opposed to invalidation) accounts for the majority of the traffic.

Figures 6.3 and 6.5 show that the traffic load increases as the object lifetime decreases. The increase is more significant for MMO than for unicast-based protocols like PET because, as the lifetime shortens, more cached copies are not referenced before being invalidated again. Volume size also affects the amount of extraneous data multicast delivered. On one hand, the web server would like to include as many objects as possible in one volume in order to amortize the multicast overhead. On the other hand, as the volume grows, the traffic load of MMO rises faster than that of unicast-based protocols (Figure 6.6). In this case, the Stanford web server should choose a volume size of 50 or less.

Despite the extraneous data, multicast-based protocols perform much better than their unicast counterparts when the number of proxies is large. Figures 6.5 and 6.6 have just 10 proxies. Figure 6.3 (500 proxies) shows that MMO outperforms other protocols even with 5-minute object lifetime. Similarly, in Figure 6.4 (500 proxies), MMO does not even reach the magnitude of AT's traffic load at volume size 100, meaning that it can carry up to 1000 objects in a volume and still outperform AT carrying 100 objects. This is because multicast scales to large audiences with little increase of traffic. For example, Figure 5.2 shows that OTERS-FT uses 24

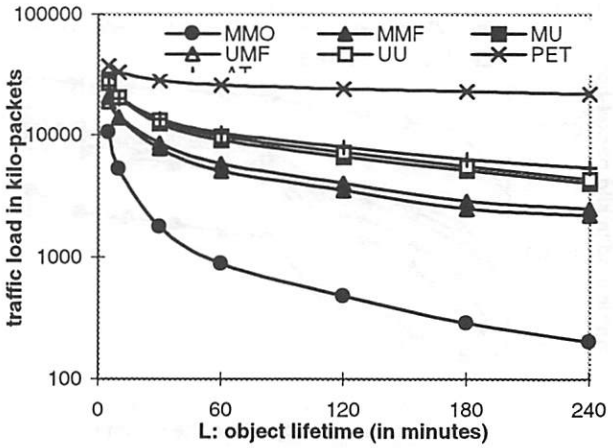


Figure 6.3 Packet Count (Surge, P=500, V=50)

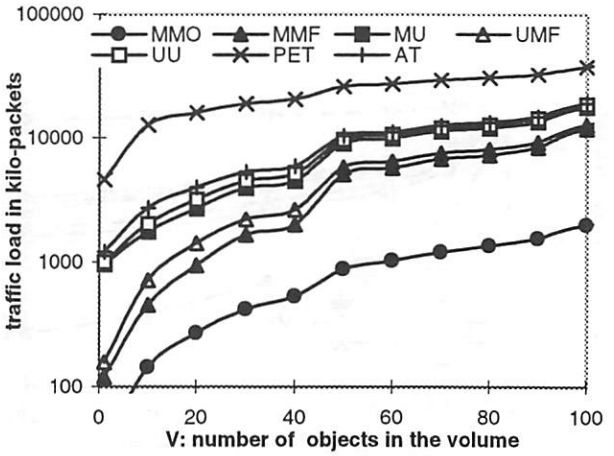


Figure 6.4 Packet Count (Surge, P=500, L=60)

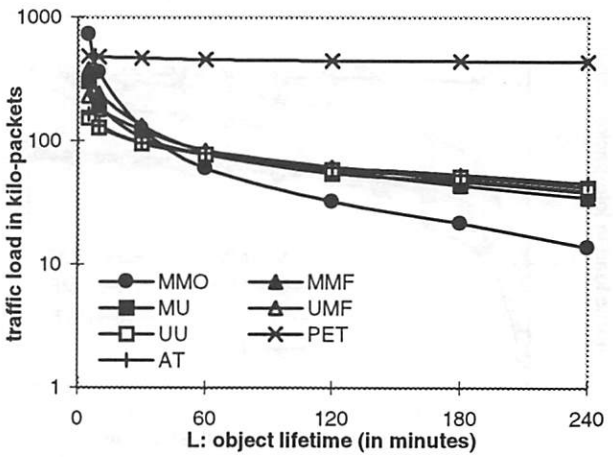


Figure 6.5 Packet Count (Stanford, P=10, V=50)

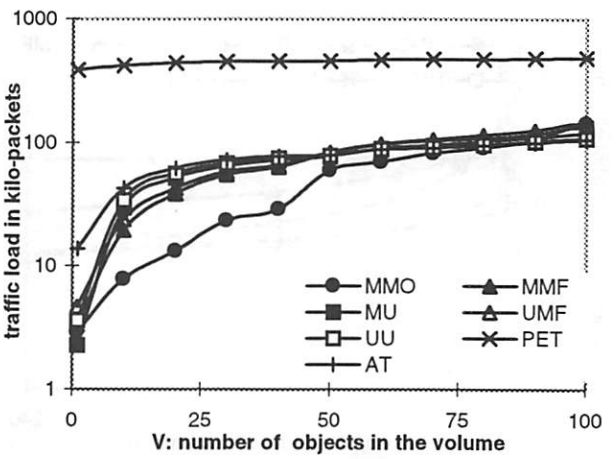


Figure 6.6 Packet Count (Stanford, P=10, L=60)

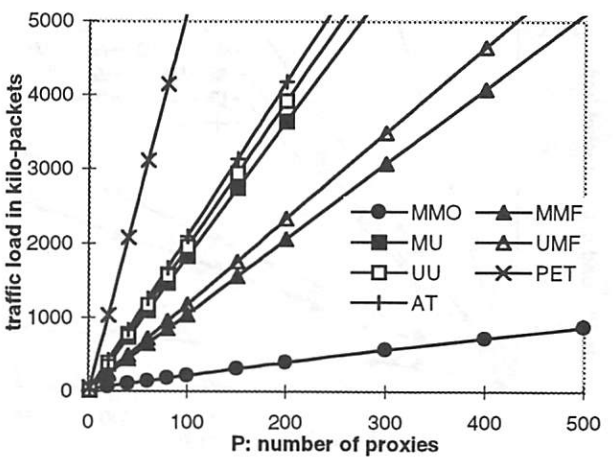


Figure 6.7 Packet Count (Surge, L=60, V=50)

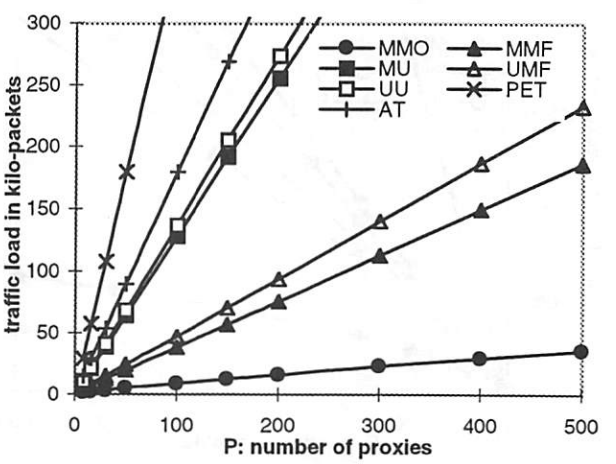


Figure 6.8 Packet Count (NLANR, L=60, V=6)

times fewer packets than TCP in order to deliver a document to 500 receivers. Therefore, MMO is less efficient than repeated AT only when over 96% of the data received is extraneous.

Figures 6.7 and 6.8 further explain the audience-size factor. Tangents of the curves follow the order:

MMO << MMF < UMF << MU < UU < AT << PET,

indicating that invalidation-based protocols are much more scalable than polling-based protocols. Moreover,

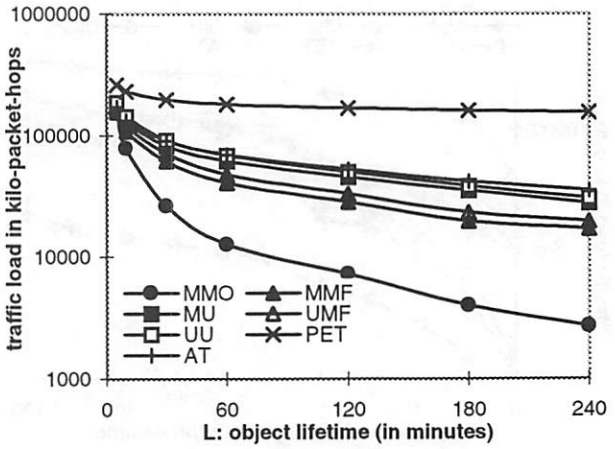


Figure 6.9 Packet-Hop Count (Surge, P=500, V=50)

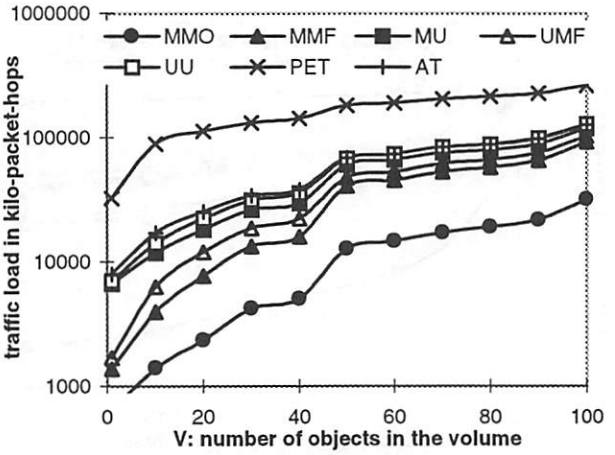


Figure 6.10 Packet-Hop Count (Surge, P=500, L=60)

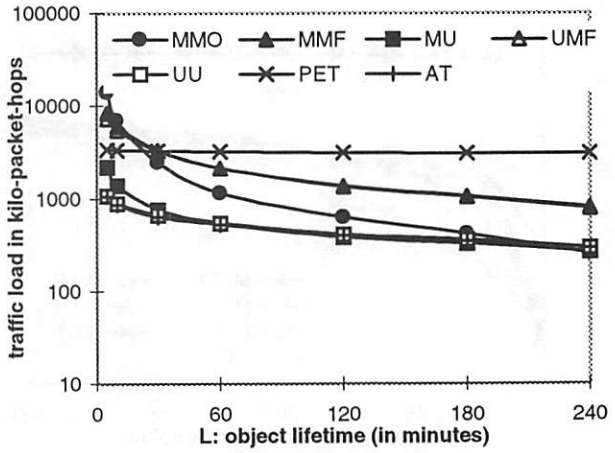


Figure 6.11 Packet-Hop Count (Stanford, P=10, V=50)

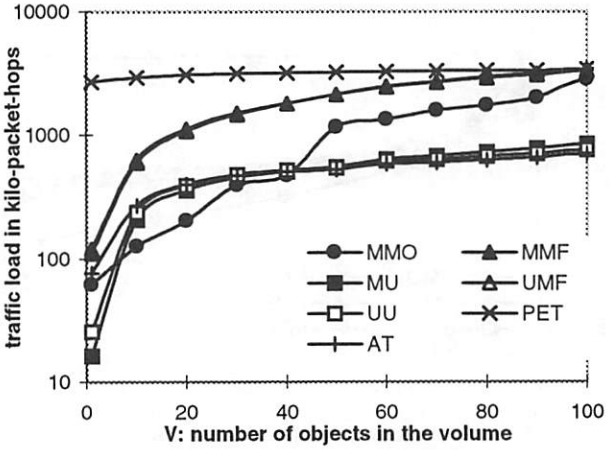


Figure 6.12 Packet-Hop Count (Stanford, P=10, L=60)

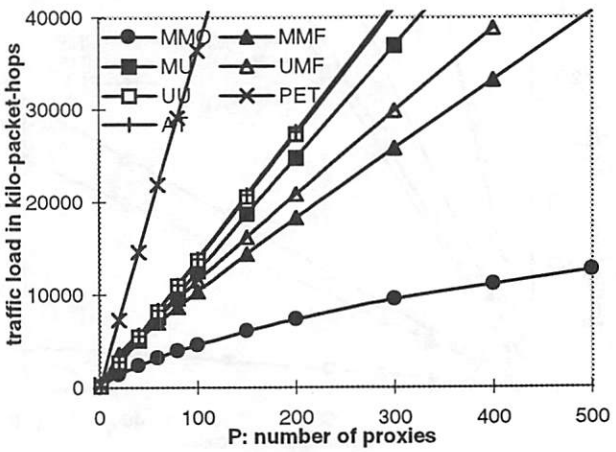


Figure 6.13 Packet-Hop Count (Surge, L=60, V=50)

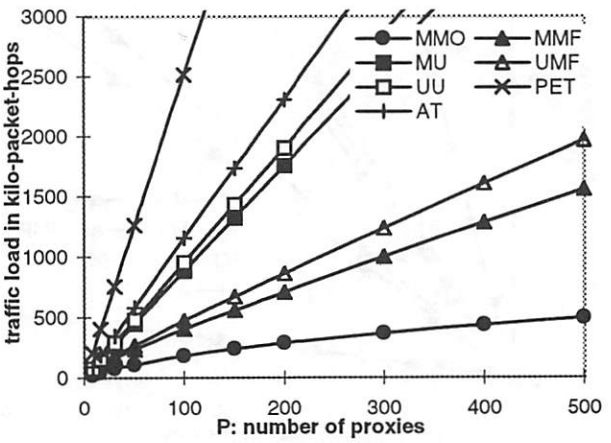


Figure 6.14 Packet-Hop Count (NLANR, L=60, V=6)

MMO is the most scalable of the invalidation-based protocols.

6.3. Network Load

Figures 6.9 through 6.14 plot the same scenarios as 6.3 to 6.8 but in packet-hop counts. Similar to the server's case, Figures 6.9 and 6.10 show that MMO is over an order of magnitude more efficient than others. With 500 proxies, multicast delivery (MMO, MMF, and UMF) always performs better than its unicast counterparts (MU and UU) and polling-based protocols (PET and

AT). Figures 6.13 and 6.14 show that MMO is far more scalable than conventional and hybrid methods from the network's perspective as well.

7. Related Work

Most related work [7, 15, 16, 29] on web caching protocols has been described in Section 3 with comparisons to MMO in that section and Section 5. Concurrent with our work, Yu et al. [37] proposed using application-level multicast for invalidations. However, their scheme presumes a pre-configured cache hierarchy in which each cache tracks web server locations and relays each HTTP miss up and down the hierarchy to the web server, and back on response. A wide flat hierarchy risks overhead from application-level routing whereas a deeper hierarchy risks latency from multiple cache hops back to the web server. This scheme, as they acknowledge, is difficult to apply with a cache mesh [3, 13, 22, 35], an emerging direction on the web. In contrast, in MMO, caches only interact as participants in a common multicast transport session; the associated subgroup hierarchy provides dynamic self-organization within a cache mesh. MMO's use of native IP multicast reduces latency and overhead on caches; its use of volumes minimizes the number of IP multicast addresses needed, addressing a key motivation of Yu [37] for going to application-level multicast. Also, the proposed EXPRESS single-source multicast [17] provides a large number of multicast addresses per server.

Another application-level technique is piggyback invalidation and validation [18,19]. However, this approach is just an optimization over unicast polling, which we have compared earlier.

Continuous multicast push (CMP) and asynchronous multicast push (AMP) [1, 26, 27] deliver popular content to end-users via native multicast. However, the server has to multicast an object continuously or many times per modification, while MMO multicasts content once per modification. Furthermore, to improve the efficiency, CMP needs to increase the amount of content carried in a multicast channel and AMP increase the wait period between two consecutive multicast deliveries, both of which prolong the end-users' web access time. Conversely, MMO reduces the web access time by always providing "fast hits" from caches.

8. Conclusion

The scalability of web caches for frequently updated objects can be significantly improved using a reliable multicast channel to proactively disseminate cache invalidations and object updates from the web server to web cache proxies. We have shown that MMO can provide fast web access, strong cache consistency, efficient bandwidth utilization and, more importantly, scalability for both the server and the network.

Considering the MMO benefits in more detail, first, the response time improves substantially for frequently updated objects (with a lifetime under 4 hours) by more than 40% over conventional caching. Second, the stale rate is reduced to zero, compared to 5% ~ 15% using a weak-consistency protocol. Even a 1% stale rate can be disastrous in applications such as medical and financial decision-making. Third, considering traffic load, MMO is over an order of magnitude more efficient than hybrid protocols, and almost two orders more than traditional ones (with 500 proxies), allowing web servers and the Internet infrastructure to meet the explosive Web growth with better service quality and lower processing and bandwidth costs.

Forming optimal volumes (so that volume objects are correlated) works better than using a separate channel for each delivery (so that proxies may choose whether or not to join the channel), in terms of reducing extraneous traffic and multicast overheads. Our experiments show that, even with random volume formation, MMO can outperform other protocols in a range of volume sizes; the range widens as the audience size grows (because of the bigger bandwidth savings over TCP). Also, the web server can form larger and better-correlated volumes based on access statistics [9]. Given a reasonably formed volume, carrying both invalidations and objects in the same channel greatly reduces the multicast session overhead as well as the address allocation and routing overhead. Conversely, our simulations find that, using a separate delivery channel, the multicast overheads can hardly be amortized over a single delivery, especially with most web objects being of small sizes.

We conclude that MMO, among the seven protocols studied, is the most efficient for disseminating popular, frequently modified and correlated objects in a volume — such as CNNfn.com or ESPN.com — to a large number of web cache proxies.

Our results to date are based on a limited set of traces. Other traces may give different quantitative results. However, we do not expect them to contradict our basic findings unless a web site hosts only highly unrelated objects.[5] The use of multicast update of cached objects

[5] The extreme is when each object is interesting to a small group of proxies and there is no overlap of interests among groups. Then no matter how the volume is constructed, either the amount of extraneous traffic is too much or the volume size and multicast group size are too small to benefit from multicast. Such objects can be disseminated via unicast.

in wide-area networks is limited in practice at present by the lack of WAN multicast support. However, as multicast is deployed in high-speed WANs to support compelling applications such as Internet TV stations, MMO is expected to become another attractive use of multicast. In fact, it completes a spectrum of delivery options for the server, from end-to-end multicast delivery for real-time video at one extreme, to multicast update of cached frequently updated objects, to unicast response to explicit requests at the other extreme. Considering this spectrum, this paper recognizes and addresses an important and growing class of objects that are less dynamic than video, yet more dynamic than can be scalably cached and kept consistent using unicast callbacks.

We hope to evaluate and refine this approach further with additional simulation and experimental deployment. One refinement is to employ delta encoding to propagate object updates [36]. In any case, our results to date indicate that this approach could play a significant role in dealing with the dramatic scaling challenges arising from the explosive growth of the Web, a growth rate that shows no sign of abating.

Acknowledgement

The authors would like to thank Paul Barford, Conrad Damon, Tim Torgenrud and the NLANR scientists for providing valuable web access traces. We also would like to thank Armando Fox, Vincent Laviano, Katia Obraczka, Craig Partridge, Shankar Ponnekanti, Chetan Rai, Jonathan Stone, and the USITS reviewers for their valuable support and comments.

References

1. Almeroth, K.C.; Ammar, M.H.; Zongming Fei; "Scalable delivery of Web pages using cyclic best-effort multicast" Proceedings IEEE INFOCOM'98 Conference on Computer Communications. April 1998. p. 1214-21 vol.3

2. Barford, P.; Crovella, M.; "Generating representative Web workloads for network and server performance evaluation" SIGMETRICS '98/PERFORMANCE'98. June 1998. Performance Evaluation Review vol.26 no.1 p. 151-60

3. Bhattacharjee, S.; Calvert, K.L.; Zegura, E.W.; "Self-organizing wide-area network caches" Proceedings IEEE INFOCOM'98 Conference on Computer Communications. April 1998. p. 600-8 vol.2

4. Bowman, C.M.; Danzig, P.B.; Hardy, D.R.; Manber, U.; Schwartz, M.F.; "The Harvest information discovery and access system" 2nd International WWW Conference. Oct. 1994. Computer Networks and ISDN Systems (Dec. 1995) vol.28, no.1-2 p.19-25

5. Byers, J. W.; Luby, M.; Mitzenmacher, M.; Rege, A.; "A digital fountain approach to reliable distribution of bulk data" ACM SIGCOMM'98 Conference. Sept. 1998. Computer Communication Review (Oct. 1998) vol.28, no.4 p. 56-67

6. Calvert, K.; Zegura, E. "GT Internetwork Topology Models (GT-ITM)" http://www.cc.gatech.edu/fac/Ellen.Zegura/gt-itm

7. Pei Cao; Chengjie Liu; "Maintaining strong cache consistency in the World Wide Web" 17th International Conference on Distributed Computing Systems. IEEE Transactions on Computers (April 1998) vol.47, no.4 p. 445-57

8. Chankhunthod, A.; Danzig, P.B.; Neerdaels, C.; Schwartz, M.F.; Worrell, K.J.; "A hierarchical Internet object cache" Proc. of USENIX Annual Technical Conference. Jan. 1996. p.153-63

9. Cohen, E.; Krishnamurthy, B.; Rexford, J.; "Improving end-to-end performance of the Web using server volumes and proxy filters" ACM SIGCOMM'98, Computer Communication Review (Oct. 1998) vol.28, no.4 p.241-53

10. Fenner, W.; Casner, S. "A "traceroute" facility for IP Multicast", Internet Draft <draft-ietf-idmr-traceroute-ipm-02.txt>, November, 1997, work in progress.

11. Floyd, S.; Jacobson, V.; Liu, C.-G.; McCanne, S.; Zhang, L.; "A reliable multicast framework for light-weight sessions and application level framing" IEEE/ACM Transactions On Networking. Dec.1997. vol.5, no.6, p. 784-803

12. Gray, C.G.; Cheriton, D.R.; "Leases: an efficient fault-tolerant mechanism for distributed file cache consistency" 12th SOSP. Operating Systems Review 1989. vol.23, no.5, p. 202-210

13. Grimm, C.; Vockler, J.-S.; Pralle, H.; "Load and traffic balancing in large scale cache meshes" TERENA Networking Conference'98. Computer Networks and ISDN Systems (30 Sept. 1998) vol.30, no.16-18 p. 1687-95

14. Gunther, R.; Levitin, L.; Schapiro, B.; Wagner, P.; "Zipfs law and the effect of ranking on probability distributions" International Journal on Theoretical Physics. Feb. 1996. vol.35, no.2, p. 395-417

15. Gwertzman, J.; Seltzer, M.; "World-Wide Web cache consistency" Proceedings of USENIX Annual Technical Conference. Jan. 1996. p. 141-51

16. Gwertzman, J.S.; Seltzer, M.; "The case for geographical push-caching" Proceedings 5th Workshop on Hot Topics in Operating Systems (HotOS-V). May 1995. p. 51-5

17. Holbrook, H.; Cheriton, D. R.; "EXPRESS Multicast: an Extended Service Model for Globally Scalable IP multicast", SIGCOMM'99, August 1999, Harvard.

18. Krishnamurthy, B.; Wills, C.E.; "Piggyback server invalidation for proxy cache coherency" 7th International World Wide Web Conference. April 1998. Computer Networks and ISDN Systems (April 1998) vol.30 no.1-7 p.185-93

19. Krishnamurthy, B.; Wills, C.E.; "Study of piggyback cache validation for proxy caches in the World-Wide Web" Proceedings of the USENIX Symposium on Internet Technologies and Systems. Dec. 1997.

20. Li, D.; Cheriton, D. R.; "OTERS (On-Tree Efficient Recovery using Subcasting): a Reliable Multicast Protocol" 6th IEEE International Conference on Network Protocols

(ICNP'98). Oct. 1998. p. 237-245

21. Luby, M. et al. "Practical Loss-Resilient Codes". Proc. of the 29th ACM Symposium on Theory of Computing, 1997.

22. Melve, I.; Slettjord, L.; Bekker, H.; Verschuren, T. "Building a Web caching system-architectural considerations" Proceedings of 8th Joint European Networking Conference(JENC8). May 1997. p. 121/1-9

23. National Lab of Applied Network Research. "A Distributed Testbed for National Information Provisioning". http://ircache.nlanr.net/Cache/

24. Padmanabhan, V.N.; Mogul, J.C.; "Using predictive prefetching to improve World Wide Web latency" ACM Computer Communication Review, July 1996. vol.26, no.3, p.22-36

25. Perkins, C. "IP Encapsulation within IP", RFC 2003, October 1996.

26. Radriguez, P.; Biersack, E.W.; "Continuous multicast push of Web documents over the Internet" IEEE NETWORK. April 1998. vol.12, no.2, p. 18-31

27. P. Rodriguez, E. W Biersack, K. W. Ross "Improving the WWW: Caching or Multicast?" 1998 Web Cache Workshop. http://wwwcache.ja.net/events/workshop/papers.html

28. Rizzo, L.; Vicisano, L.; "A reliable multicast data distribution protocol based on software FEC techniques" Proceedings of Fourth Workshop on the Architecture and Implementation of High Performance Communications Subsystems - HPCC'97. June 1997. p. 115-24

29. Touch, J. "The LSAM Proxy Cache — a Multicast Distributed Virtual Cache" 1998 Web Cache Workshop. June 1998. http://wwwcache.ja.net/events/workshop/14/lsam.html

30. UCB/LBNL/VINT Network Simulator - ns (version 2), http://www-mash.cs.berkeley.edu/ns/

31. D. Waitzman, C. Partridge and S.E. Deering, "Distance Vector Multicast Routing Protocol", RFC1075, Nov. 1988.

32. Wessels, D.; Claffy, K.; "ICP and the Squid web cache" IEEE Journal on Selected Areas in Communications. April 1998. vol.16, no.3, p. 345-57

33. Yin, J.; Alvisi, L.; Dahlin, M.; Lin, C.; "Using leases to support server-driven consistency in large-scale systems" Proceedings of 18th International Conference on Distributed Computing Systems. May 1998. p. 285-94

34. Yu, P.S.; MacNair, E.A.; "Performance study of a collaborative method for hierarchical caching in proxy servers" 7th International World Wide Web Conference. April 1998. Computer Networks and ISDN Systems (April 1998) vol.30 no.1-7 p.215-24

35. L. Zhang, S. Michel, K. Nguyen, A. Rosenstein "Adaptive Web Caching: Toward a New Global Caching Architecture" 1998 Web Cache Workshop, http://wwwcache.ja.net/events/workshop/25/3w3.html

36. Mogul, J.C.; Douglis, F.; Feldmann, A.; Krishnamurthy, B.: "Potential benefits of delta encoding and data compression for HTTP" ACM SIGCOMM 97 Conference. Computer Communication Review (Oct. 1997) vol.27, no.4 p. 181-94

37. Haobo Yu, Lee Breslau, and Scott Shenker, "A Scalable Web Cache Consistency Architecture" ACM SIGCOMM'99

Appendix A. the Web Access Traces

Surge [2] generates 500 proxy traces. Each aggregates requests from 2000 clients and lasts 15 hours. So the trace covers one million web clients. Requests are generated for 100 frequently updated objects (called *hot* objects). The number of requests for a hot object and its popularity rank follow the *zipfs law* [14]. The most popular object is accessed an average of 0.5 time per client, which is fairly conservative for web sites like CNN.com. In other words, a proxy receives 1000 requests to the most popular object and in total 6200 requests to the 100 hot objects. File sizes follow a hybrid Pareto and log-normal distribution with average 8.6 KB, standard deviation 85 KB, minimum 79 bytes and maximum 858 KB.

The Stanford trace is a 24-hour server log on December 8, 1998. After filtering out non-cacheable requests, the log contains 960,548 requests made by 42,804 clients to 97,630 files. 1% of the files are picked (uniformly across the popularity ranks) as hot objects. Popularity ranks are obtained by sorting the files based on the number of requests each file receives. Then out of every 100 files (consecutive on the sorted list), one is picked randomly as a hot object. The most popular object is accessed 51,687 times. The average file size is 24.6 KB with 1 byte minimum and 225 MB maximum. Clients are randomly partitioned into 10 groups. Requests from one group of clients form one proxy trace. The server trace is thus partitioned into 10 proxy traces.

The NLANR trace [23] consists of eight 24-hour proxy traces on December 17, 1998, the first day of the Desert Fox US military operation against Iraq. We selected CNN.com as the server site and a volume based on the prefix "*http://www.CNN.com/WORLD/meast/9812/17/iraq.strike*". There are 6 objects in the volume with average size 20.7 KB, standard deviation 8.4 KB, minimum 8.1 KB and maximum 30.7 KB. In the simulation, we scale up the number of proxies by replicating the traces.

Table A.1 shows that each NLANR proxy does not have many clients and requests to CNN.com. This is because these proxies are at the top of the *NLANR Cache Hierarchy*, each covering domains like .uk and .jp. Their clients are mostly lower-level web cache proxies. Hence the request streams are already highly filtered and reduced. Nevertheless, they represent an important part of the web caching reality.

Only the NLANR trace records the proxy response time (the time between reading the first byte of the request

Proxy name:	bo1	bo2	lj	pa	pb	sd	sv	uc
# clients:	55	57	49	39	55	36	65	48
total # requests to CNN:	2,191	2,402	3,317	4,200	6,686	7,668	5,093	2,833
# requests in the volume:	258	301	228	435	521	240	441	315

Table A.1 The number of clients and requests for NLANR proxies

Proxy name:	bo1	bo2	lj	pa	pb	sd	sv	uc
response time of a fast hit:	35	18	58	75	104	127	183	75
response time of a slow hit:	286	215	306	271	440	408	601	292
response time of a miss:	632	564	839	978	971	1,216	950	682

Table A.2 The average response time of NLANR proxies (in milliseconds).

and writing the last byte of the reply) and whether the response is a fast hit (TCP_HIT), a slow hit (REFRESH_HIT) or a miss (REFRESH_MISS). Note that TCP_MISS (a miss in the proxy's cache) does not occur in the simulation other than at the first time because any object in the volume is cached once accessed. Samples larger than 2 seconds are discarded. For each object, response times of multiple requests are averaged into one value for each response type, which the simulation then uses. Table A.2 has the response times further averaged across all objects in the volume. It shows that a fast hit offers far better response time than a slow hit or miss.

Appendix B. the Transport Protocols

TCP, OTERS and Digital Fountain are simulated on NS [30]. Ten transit-stub topologies[6] are generated by the GT-ITM internetwork topology generator [6]. Each topology has 1000 nodes, including 5 transit domains and 120 stub domains. Nodes are considered as either backbone routers or web cache proxies at the borders of their respective local area networks. Behind each node there may be hundreds or thousands of hosts that use the web caching service and are connected via ISP networks or corporate LANs. Links inside a stub domain are 100Mbps with 1ms delay. Links connecting stub and transit domains are 45Mbps with 15ms delay. Links inside a transit domain are 155Mbps with 8ms delay. Inter-transit-domain links are 155Mbps with 80ms delay. Link delays have random variations that adhere to an unbounded exponential distribution with 20% average variation. Losses are random and the link packet error rate (PER) is 1%. The multicast routing is static dense mode DVMRP [31].

For a given multicast group size, the following occurs. Each protocol's simulation is run 10 times on each of the 10 topologies with different seeds. The seed controls how receivers are randomly chosen from the 1000 nodes. Results of the 100 runs are averaged to produce the protocol's packet count and packet-hop count. Linear interpolation is used to estimate the traffic load of group sizes that are not simulated. Finally all the results are plotted in Figures 5.1 through 5.4 and fed to the web-caching simulation. The simulation is driven by a packet stream from a Constant Bit Rate (CBR) source at 100 KB/sec and 1KB packet size. In TCP, the CBR source reliably unicasts a file to each receiver. Its traffic load counts in all the data and ACK packets. Favoring TCP, we assume the connection setup and termination takes 5 packets rather than 7.

In OTERS, all the receivers join a multicast group at time 0 and start organizing the subgroup hierarchy (called the *fusion tree*). Whenever there is data in transmission, each group member sends heartbeats every 1 second (with random skews) to maintain the tree. The data transmission starts at 0.2 second, when the fusion tree is only partially formed. (This overlap causes extra reliability cost while the tree is under construction, which can happen when group members join and leave.) One notification is delivered using OTERS-NT, followed by a file transfer using OTERS-FT. The session overhead comes from the session organization packets. The per-packet cost consists of heartbeats, ACKs, NAKs, and retransmissions, along with the payload data. We also tried to extract *per-file overhead* but it turns out to be under 1% of the per-packet cost and therefore is not considered as a separate term in the traffic load model.

We simulated the Digital Fountain designed by Byers et al. [5], which uses Tornado codes [21] with $n = 2k_1$. The data transmission starts at time 0. All the receivers join the multicast group at time 0 and leave as soon as k_2 packets are received. According to [5], the average of k_2 is tuned to 5.48% over k_1. With 1% link PER, no receiver experiences over 50% losses (which is largely the case for any well connected web cache). Therefore the source always needs to send only two packets per payload packet. Packet-hop-wise, a router may continue to forward packets toward a receiver after it has left the group, until its multicast leave message reaches the router. To separate this overhead from the per-packet cost, multiple sets of results are collected for different file lengths. The session overhead and per-packet cost are then extracted from them.

[6] The picture at ftp://ftp.dsg.stanford.edu/pub/papers/ts0.gif shows an example 100-node topology. The ones used are 10 times larger.

Hierarchical Cache Consistency in a WAN*

Jian Yin, Lorenzo Alvisi, Mike Dahlin, Calvin Lin
Department of Computer Sciences
University of Texas at Austin
yin,lorenzo,dahlin,lin@cs.utexas.edu

Abstract

This paper explores ways to provide improved consistency for Internet applications that scale to millions of clients. We make four contributions. First, we identify how workloads affect the scalability of cache consistency algorithms. Second, we define two primitive mechanisms, *split* and *join*, for growing and shrinking consistency hierarchies, and we present a simple mechanism for implementing them. Third, we describe and evaluate policies for using split and join to address the fault tolerance and performance challenges of consistency hierarchies. Fourth, using synthetic workload and trace-based simulation, we compare various algorithms for maintaining strong consistency in a range of hierarchy configurations. Our results indicate that a promising configuration for providing strong consistency in a WAN is a two-level consistency hierarchy where servers and proxies work to maintain consistency for data cached at clients. Specifically, by adapting to clients' access patterns, two-level hierarchies reduce the read latency for demanding workloads without introducing excessive overhead for nondemanding workloads. Also, they can improve scalability by orders of magnitude. Furthermore, this configuration is easy to deploy by augmenting proxies, and it allows invalidation messages to traverse firewalls.

1 Introduction

To improve performance and reduce bandwidth, caching has become a ubiquitous Internet technology. However, web caching introduces the problem of maintaining consistency. With weak notions of consistency users can observe confusing data, and innovative web services—such as agents, robots, and distributed databases—will likely produce incorrect results. Furthermore, consistency polling can increase server load, increase latency, and reduce the effectiveness of large-scale caches [9, 5]. Thus, improved consistency will become increasingly desirable.

*This work was supported in part by an NSF Research Infrastructure Award CDA-9624082, DARPA/SPAWAR grant number N66001-98-8911, and grants from Dell, Novell, and Sun. Dahlin and Alvisi were also supported by NSF CAREER grants CCR-9733842 and CCR-9734185, respectively.

Callback-based consistency can be used either to provide strong consistency—which guarantees that a client's read of an object returns the latest completed write of that object—or best effort consistency—which attempts to invalidate stale data when it changes and which can limit the amount of time during which a client unknowingly accesses stale data [20]. However, simple callbacks are unacceptable for a WAN because servers may be forced to delay their writes indefinitely when there are client failures or network partitions.

In previous work, we show how to combine callbacks with timely server writes by using *Volume Leases* [20], a generalization of the original notion of *leases* [7]. Volume Leases are a fundamental abstraction that provides a mechanism for enforcing strong consistency semantics that is separate from the policy question of when to renew volume leases. We focus on a policy where clients fetch volume lease renewals on demand when they access a volume. We plan to explore other policies such as prefetching or multicasting lease renewals in the future. Our earlier study uses trace-driven simulations and show that Volume Leases perform well compared to object leases and polling.

The key questions that this paper answers are (1) how large a system can Volume Leases accommodate, and (2) what techniques can be used to scale to even larger systems? To answer these questions, this paper explores ways to accommodate popular web servers with millions of clients by combining Volume Leases with hierarchies.

Adding hierarchy to callback-driven consistency can yield three benefits. First, latency can be reduced if clients can register callbacks or renew leases by going to a nearby node in the consistency hierarchy rather than to the server. Second, hierarchy can improve network efficiency by forming a multicast tree for sending invalidation messages to caches and a reduction tree for gathering their replies. Third, hierarchy improves server scalability by distributing load and callback state across a collection of nodes.

However, using hierarchy for scalable consistency introduces its own challenges. Availability may suffer because hierarchical structures consist of multiple nodes

that can fail independently. Also, latency can increase if the hierarchy must be traversed to satisfy requests. Finally, it is unclear how best to configure the hierarchy.

This paper develops solutions for hierarchical consistency and addresses these issues. We make four contributions. First, we identify and quantify the ways in which specific characteristics of data-access workloads affect the scalability of cache consistency algorithms. Second, we define two primitive mechanisms, *split* and *join*, for growing and shrinking hierarchies, and we show how these primitives can be implemented with a simple mechanism already present in the Volume Leases algorithms. Third, we describe and evaluate policies for using split and join to address the fault tolerance and performance challenges of hierarchies. Fourth, we examine and compare algorithms for maintaining consistency in a range of hierarchy configurations.

We explore three configurations. First, in *generic hierarchy*, consistency proxies can be placed anywhere in the Internet. The second configuration, *server-proxy-client*, is designed to exploit widely deployed web proxies, which serve as gateways between enterprise LAN and web servers to improve security and network efficiency. In this configuration, web proxies are augmented to serve as consistency proxies that forward invalidation messages from web servers. This addresses the engineering challenge introduced by firewalls that generally prevent external machines from sending invalidation messages directly to clients within the firewall. The third configuration is the *server cluster* configuration in which hierarchy is introduced only within a LAN cluster of servers to improve scalability.

We evaluate our algorithms using simulation. To study scalability and to evaluate how the system is affected by different workload characteristics, we first use a series of synthetic workloads. To calibrate these results with realistic workloads, we also examine some smaller trace-based workloads. Overall, we find that even without hierarchies, Volume Leases can scale to services with tens of thousands of clients; with hierarchies, scalability beyond millions of clients appears feasible.

The thesis of this paper is not that all servers should provide strong consistency, but rather that for Internet-scale systems, strong consistency is feasible for a wide range of applications. We envision flexible systems where either servers or clients can specify the consistency semantics for their data [10]. The algorithms described here can also be used to provide *best effort consistency* where servers make a best effort to notify clients of changes to cached data, but servers do not delay writes. We discuss best effort versions of Volume Lease consistency algorithms in detail elsewhere [20].

The rest of this paper proceeds as follows. Section 2 discusses a few ideas that are needed to understand this work. Section 3 describes our new algorithm whose performance we evaluate in Section 4. We close by discussing related work and drawing conclusions.

2 Background

This section describes four concepts necessary to understand hierarchical consistency: callbacks, leases, the Volume Leases algorithm, and reconnection under Volume Leases.

In server-driven consistency, a client registers *callbacks* with a server for objects that it caches [9, 15]. Before modifying an object, a server sends invalidation messages to all clients that have registered interest in that object. The advantage of this approach is that servers have enough information to know exactly when cache objects must be invalidated. By contrast, in client-driven consistency schemes, such as those currently used in NFS and HTTP, clients periodically ask the server if objects have been modified. This creates a dilemma for the client. A short polling period increases both server load and client latency, while a long polling period increases the risk of reading stale cache data.

However, there are two challenges for server-driven consistency in large distributed systems. First, scalability is an issue, because large numbers of clients lead to large server state and large bursts of load when popular objects are modified. Second, performance in the face of failures is an issue because servers cannot modify an object until all clients have been notified that their cached copies are no longer valid. Because of this requirement, server writes can be delayed indefinitely while the server waits for acknowledgments from unreachable clients.

These challenges can be addressed by introducing *leases* [7]. When a client registers a lease with a server, the lease specifies some time T during which the server will notify clients of updates. Leases improve scalability because servers need to track only active clients, and they improve fault tolerance because even if a client is unreachable, writes are delayed only until the client's lease expires. The lease length T presents a trade-off. Long leases minimize the overhead of renewing leases, while short leases reduce server state and improve failure-mode write performance.

Leases do not perform well for web workloads because the interval between a client's reads is typically long, so object leases must be long to amortize the cost of lease renewals across multiple reads [20]. The *Volume Leases* algorithm introduces the notion of a volume, which is a collection of objects that reside on the same server. The algorithm associates a lease with each volume as well as with each object. A client's cached object is valid only if both its object lease and corresponding volume lease are valid. The Volume Leases algorithm uses a combination of long object leases and short

volume leases to resolve the tradeoff with lease lengths. Short volume leases allow servers to write quickly in the face of client and network failures: since clients can't read an object when its corresponding volume lease is invalid, in the worst case the server waits only for the the short volume lease to expire before modifying an object. While the cost of renewing short volume leases is amortized across the number of objects that reside in the same volume, long object leases minimize the overhead of renewing object leases.

In the Volume Leases algorithm, a server maintains a list of unreachable clients whose volume leases expired while the server was attempting to invalidate an object lease. We call this list the *unreachable list*. After an unreachable client recovers or its network connection to the server is restored, the next time that client tries to renew its volume lease the algorithm uses a reconnection protocol to restore consistency between the client's and server's lists of current object leases.

Because the reconnection protocol is a key building block for hierarchical caching, we describe it in detail. Each server maintains an *epoch number*. Whenever a server recovers from a crash, it increments the epoch number and logs that number to stable storage before proceeding with normal operations. All messages from a server to its clients include the epoch number. When a client receives a message from a server, it records the server's epoch number. When a client sends a volume lease request to a server, it always includes the last known epoch number for that server. A server grants a volume lease only if the epoch number in the request matches its current epoch and if it has not marked client unreachable. If the client's epoch number does not match or if the client is marked unreachable, the server sends the client a reconnect request. In response to such a request, a client sends the server the list of objects it currently caches and the *version numbers* of these objects. The version number associated with an object is an integer that the server increments whenever it modifies the object. The server then compares the version numbers of the cached objects and server's objects and grants object leases to all objects whose versions match. The server invalidates all other cached objects then grants the volume lease. All these tasks can be accomplished with one message from the server to the clients. When the client finishes updating its object leases, it sends a connect message back to the server, which then removes the client from the unreachable list.

3 Algorithms

We first describe a naive algorithm based on a static consistency hierarchy and discuss its performance and fault tolerance properties. Next, we present two primitive mechanisms, split and join, for reconfiguring the hierarchy. These mechanisms can be constructed with trivial additions to the basic Volume Leases algorithm. We then describe policies that use these mechanisms to enhance the fault tolerance and performance of the basic static hierarchy.

Both the static and dynamic versions of the algorithm assume that nodes participating in the consistency service have been identified and organized into an initial hierarchy. This study does not specify a particular mechanism for doing so. For some systems, constructing the hierarchy manually suffices; for some, such as the server-proxy-client configuration in Section 4.3, automatic construction is trivial; for others, more sophisticated automatic strategies such as those described by Plaxton et. al [16] might be required. This hierarchy may be embedded on current clients and proxies, it might be coincident with a larger cache hierarchy [2] or it might be part of a separate data-location-metadata hierarchy [6, 18].

3.1 Static hierarchy

Our consistency hierarchy is a tree structure of interconnected nodes. We refer to the root as the *origin server*, to the leaves as *clients*, and to the intermediate nodes as *consistency servers*. Each node runs the standard Volume Leases algorithm; each intermediate node acts both as a client and as a server, treating its parent as its server and its children as its clients. Each node thus satisfies lease requests from its children by returning a valid lease if it has one cached, or—if it does not—by requesting a lease from its parent, caching the lease, and returning the lease to its child. Similarly, each node passes to any children that have valid leases the invalidation messages that it receives.

Such hierarchies have the potential to improve performance by reducing both server load and the latency of client lease renewals. In the Internet, a popular site might serve millions of clients, and by using a hierarchy, a server tracks and communicates with only its immediate children. This reduces memory state, average load for lease renewals, and bursts of load when popular objects are modified. In essence, the consistency hierarchy forms a multicast tree for sending invalidation message and forms a reduction tree for gathering replies. By the same token, if clients can renew leases by going to nearby intermediate consistency servers rather than to the root server, read latency and network load may be reduced.

However, the use of leases in the hierarchy is not guaranteed to reduce either server load or latency. When volumes are popular and frequently accessed, it is likely that consistency servers will hold valid leases and will respond to client requests without consulting their parents, and it is likely that the hierarchical "multicast" will achieve a large fan-out and significantly reduce server load. However, for unpopular or infrequently accessed

volumes, the time between accesses to consistency nodes is likely to be longer than the volume lease, so the cached leases may often have expired when they are accessed. In these cases, many messages would traverse the entire hierarchy, increasing the average read latency without reducing server load.

A second problem with a static hierarchy is reliability. The hierarchy consists of a large number of nodes that can fail independently, and one node failure can effectively disconnect a subtree.

3.2 Join and split

The solution to both problems is to reconfigure the consistency hierarchy dynamically without breaking consistency guarantees. We propose a mechanism that uses two primitives: *join*, which removes an intermediate node from the hierarchy, and *split*, which adds an intermediate node to the hierarchy. Both primitives work on a per-volume basis—in our system different volumes can use different hierarchies.

Join and split can be trivially implemented using mechanisms already required by the Volume Leases algorithm. Recall that join removes a node from the hierarchy, connecting the children of the node directly to the node's parent. To implement join we augment the volume epoch number to include the parent node's identity. When a child initiates a join for a particular volume, it simply begins using its former grandparent as a parent. The volume epoch number held by the child will not match its new parent, so the new parent initiates the standard volume reconnection protocol to synchronize its state with its new child. Thus, going to a new parent in the hierarchical algorithm is no different than going to a server that has crashed and lost a client's state in the original Volume Leases algorithm. Similarly, to split the hierarchy, a child chooses a descendant of its parent and starts using the new node as its parent, again using the reconnection protocol to synchronize the state. For both split and join, the decision to use a new parent can be made by children at arbitrary times. The criteria for such decisions are a matter of policy. Children can thus decide to find new parents to improve fault tolerance or they can be told to use new parents to improve performance.

3.3 Fault tolerant static hierarchy

Using join and split, an intermediate node failure is handled as follows. If a node N cannot contact its parent P to renew a lease, it sends the renewal message to one of its ancestors A, triggering the volume reconnection protocol between N and A. Note that if A cannot send an invalidation to P, it does not try to contact N, but instead waits for the volume lease timeout; this means that parents need to know only about their immediate children, not their more distant descendents. Finally, when node P recovers, it can send hints to its list of (former) children suggesting that they split from A and join P instead.

3.4 Dynamic hierarchy configuration

For volumes with high read frequencies and many active clients, a deep hierarchy can reduce read latency and distribute load. However, for less popular objects, or for popular objects with low read frequency, intermediate hops can increase read latency without significantly reducing server load. Therefore, it is useful for different volumes to construct different dynamic hierarchies. These hierarchies can be constructed from the static hierarchy using the split and join mechanisms in response to changing workloads. Hence, a node can have different children in the static and dynamic hierarchies: we refer to the former as *static children*, and to the latter simply as *children*.

In the dynamic configuration algorithm a node N monitors the number of lease requests it receives from its children and the fraction of these requests that it can satisfy locally during time intervals of length T. Using this data, N instructs its children to join with its parent if (1) the load from its children would not cause the load on its parent to exceed a threshold value and (2) its children would receive better read latency by skipping N and going directly to the parent. N performs the latency calculation as follows.

Let $RenewCost(N)$ be the cost for a child of N to renew a lease cached at N, and let $RenewCost(P)$ be the cost for N to renew a lease cached at its parent. If the fraction of renewals that N satisfies locally is F, then the expected latency that a child of N pays to renew a lease is $RenewCost(N) + (1-F)RenewCost(P)$. Assuming that the cost of accessing N's parent is about the same for both N and N's child, the expected cost after a join is $RenewCost(P)$. When $RenewCost(N) + (1-F)RenewCost(P)$ is greater than $RenewCost(P)$ by some threshold, N instructs its children to perform a join unless doing so would raise the load of the parent to an unacceptable level.

Similarly, to determine when to initiate a split, a node monitors the requests from its children and initiates a split if (1) its local load exceeds some threshold or (2) connecting a set of children to a skipped node would reduce their expected read latency by some threshold.

A node N performs this read latency calculation by simulating the performance of its skipped children as follows. For each static child S, N maintains a simulated request count $ReqCount(S)$, hit count $HitCount(S)$, and volume lease expiration time $VolExp(S)$. When a child C of N contacts N to renew a lease, N updates the statistics for the skipped child S that is a static ancestor of C by (1) incrementing $ReqCount(S)$, (2) incrementing $HitCount(S)$ if the current time is before $VolExp(S)$, and (3) setting $VolExp(S)$ to

the current time plus the volume lease length. Let $RenewCost(N)$ be the cost for C to renew its lease at N and $RenewCost(S)$ be the cost for C to renew its lease from the skipped child S instead. N tells C and its siblings to split from N and instead use S as their parent if $RenewCost(S) + (1 - \frac{HitCount(S)}{ReqCount(S)}) \cdot RenewCost(N) < RenewCost(N) - threshold$.

4 Evaluation

We evaluate hierarchical consistency in three different deployment configurations. First, we examine an aggressive deployment model, *generic hierarchy*, to characterize the factors that affect the behavior of the core algorithms and to determine the performance limits of our approach. Second, we examine a simple *clustered-server* configuration in which the hierarchy is used to distribute the algorithm across a LAN cluster in order to improve scalability but not latency. This configuration might be used if a service wishes to provide strong consistency for its data without relying on having consistency-enabled intermediate proxies deployed across the WAN. Third, we examine a *server-proxy-client* configuration that maps well to infrastructure that is common today.

We evaluate these algorithms using simulations. To study scalability and to evaluate how different aspects of workloads impact scalability, we first use a series of synthetic workloads. We run each of these experiments five times using different random seeds for workload generation and show the 90% confidence interval for each point. Then, to calibrate these results, we examine a smaller, trace-based workload in the context of the server-proxy-client configuration.

Based on these experiments, we reach the following conclusions:

- For the aggressive deployment scenario with flexible hierarchy configurations, static hierarchies can reduce latency compared to the flat Volume Leases algorithm for high request-rate services, but they can increase latency for low request-rate services. In contrast, the dynamic version always performs as well as the flat algorithm for low request rates and as well as the static hierarchy for high request rates.

- For workloads with modest request rates in the range of many current web services, the flat Volume Leases algorithm with a single server can scale to client populations in the tens or hundreds of thousands of nodes; distributing the consistency algorithm across a group of nodes—either in a cluster or across a WAN—via hierarchies can provide scalability to millions of clients even under aggressive workloads.

- In the server-proxy-client configuration, the simple static hierarchy performs well for our web trace workload; this configuration has the added benefit that it might also provide a controlled way to traverse firewalls in order to deliver consistency signals. The synthetic workload suggests that there may be other workloads for which the dynamic algorithm's flexibility is desirable.

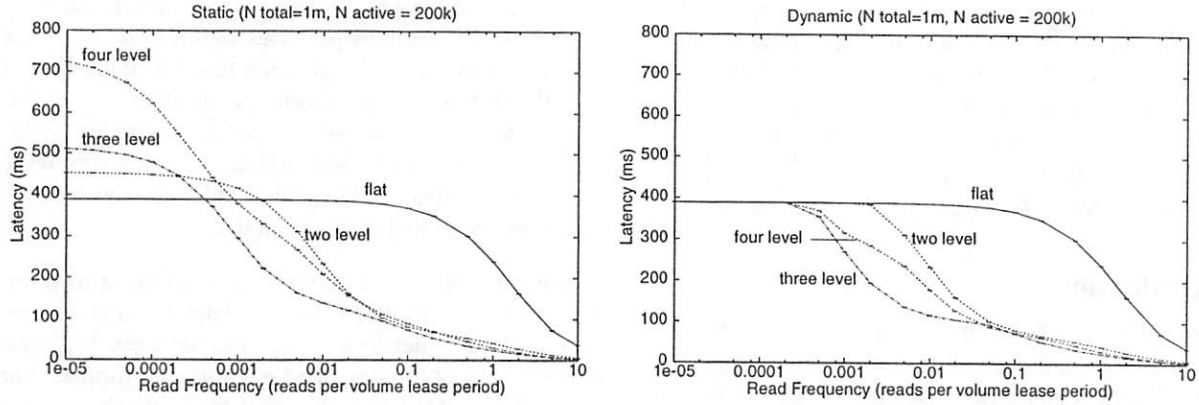
Our methodology makes several significant assumptions and simplifications. For our latency estimates, we do not simulate network or server contention. We use a simple network topology and delay model to make our analysis tractable. Finally, our default synthetic workloads simulate one object per volume. This may understate the apparent benefit of hierarchies because long-lived object leases are much easier to cache in the hierarchy than short volume leases; furthermore, the small number of objects per volume may also hurt the relative performance of the static algorithm.

4.1 Generic hierarchy

Our Generic Hierarchy configuration represents a system with few constraints on deployment. We examine this configuration to understand the behavior of the core algorithms as we vary several key parameters. This configuration also models an aggressive deployment strategy such as might be employed within a large cache service or in a system where collections of servers and cache systems coordinate to provide consistency.

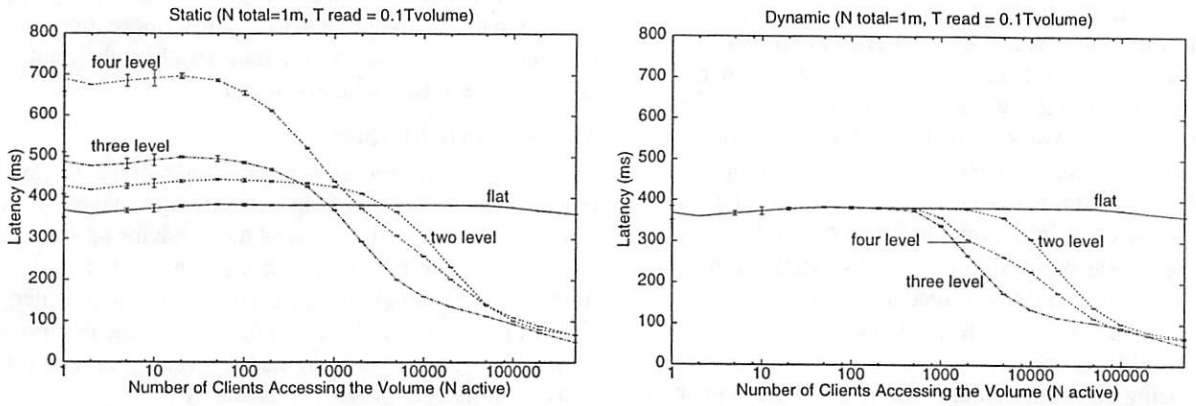
The consistency hierarchy is a tree with one server at its root, C clients at its leaves, and $l - 1$ levels of intermediate nodes. We designate the server to be the level 0 node of the consistency hierarchy. For simplicity, we assume that at all levels of the tree the degree d is the same, with $d^l = C$. We defer the evaluation of hierarchies with different fan-out at different levels for future work. We use a simple cost model for accessing consistency servers. First, we assume that all leaf nodes and internal nodes within a subtree experience the same latency when they renew a lease with the root of that subtree. Second, we assume that the latency experienced within a subtree increases with the number of leaves in the subtree as follows: subtrees with 100 or fewer leaves have a latency of 30 ms, subtrees with 10,000 leaves have a latency of 100 ms, and subtrees with more than 100,000 leaves have a latency of 400 ms; latencies for subtrees with 100-10,000 nodes and 10,000 to 100,000 nodes are estimated through interpolation. These latencies are meant to be suggestive of department-, enterprise-, and Internet-scale delays, but do not represent any specific system.

We use a synthetic workload and compute the average read latency and server load when we simulate the accesses of a collection of clients to a single volume. Out of N_{total} clients, we choose a subset of clients of



(a) Static (b) Dynamic

Figure 1: Average read latency as the per-client read frequency is varied for a hierarchy of one million clients, of which 200,000 access the volume in question.



(a) Static (b) Dynamic

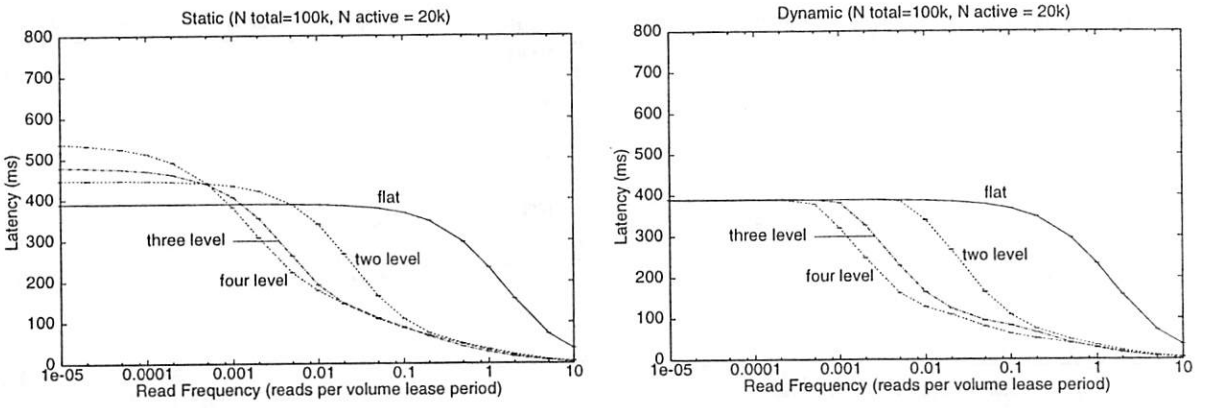
Figure 2: Average read latency as the number of active clients varies for a hierarchy of one million clients, each issuing requests to a volume at a rate of 0.1 requests per volume lease period.

size N_{active} that access the volume with per-client inter-access times determined using an exponential distribution around an average value t_{read}, which is expressed as a ratio of the average inter-access time to the volume lease renewal time. In our initial experiment, each volume contains a single object; we relax this assumption later in this section.

Read Frequency A server's read frequency has a large impact on the performance of hierarchical consistency. The higher the collective read frequency of the clients below a proxy, the more often the proxy holds the lease. Hence, if the read frequency is high the lease hit ratio will be high and the proxy can reduce read latency. Otherwise, if the collective read frequency is low, then the lease hit ratio will be low.

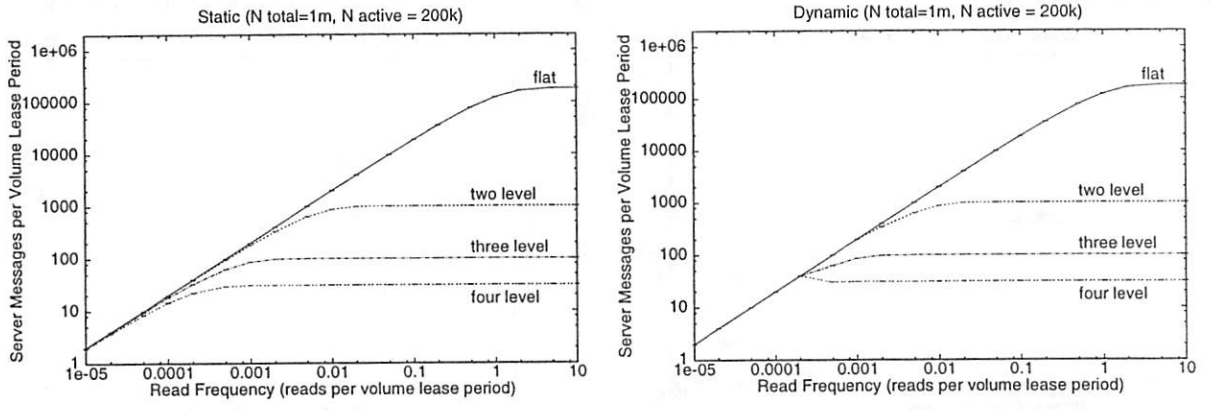
Figure 1 shows the average lease renewal latency as the per-client read frequency is varied. Figure 2 shows lease renewal latency as the fraction of clients that access the volume in question is varied. The graphs compare the performance of a flat, 2-level, 3-level, and 4-level consistency hierarchy with part (a) of each figure showing performance for the static algorithm and part (b) showing performance for the dynamic algorithm. Figures 1 and 2 have the same general shape because as one moves to the right along the x axis the total request rate increases in both sets of graphs. But, these graphs represent different dimensions of the design space. Read latency generally decreases as read frequency increases. For high request rates, the read latency falls even for the flat configuration because a client issues multiple reads within the period the client's volume lease is valid.

To interpret these graphs it is helpful to consider where different classes of services might lie or where a single service might lie under different workloads. For example, a weather service that is visited by an average client once a day for one minute and that uses a 10-second lease period would correspond to a read frequency of less than 0.001 reads per volume lease period per client. Similarly, a news service whose typical users visit for 5 minutes during the 8-hour working day would correspond to a volume renewal frequency near 0.01 per volume lease period per client. The read frequency of that same ser-



(a) Static (b) Dynamic

Figure 3: Average read latency as the per-client read frequency is varied for a hierarchy of 100,000, of which 20,000 access the volume in question.



(a) Static (b) Dynamic

Figure 4: Average server load for handling renewal requests as the per-client read frequency is varied for a hierarchy of one million clients, of which 200,000 access the volume in question.

vice might jump above 0.1 or even near 1 for periods of time during news events of widespread interest as clients constantly monitor the news for new developments. Similarly, emerging program-driven applications might span a wide range of the parameter space.

With respect to lease renewal latency as a function of read frequency, the main observations are as follows:

- Hierarchies can significantly reduce latency for active and popular services.
- The dynamic hierarchy succeeds in matching the latency of the flat Volume Leases algorithm for less active or less popular services while matching the performance of the static hierarchy for busier services. Relative to flat Volume Leases, the static hierarchy can hurt latency for less active or less popular services but can help latency for active and popular services.
- The dynamic hierarchy appears to be a good default choice for this configuration. If a service's access patterns are known and if these access patterns do not change much, then either flat Volume Leases or a static hierarchy may be reasonable, depending on the workload.

Finally, note that the variations among different depths of underlying static trees depend both on interactions between the number of clients under each level of a subtree and on our assumptions on the network distances between subtrees as a function of subtree size. Hence, this experiment should not be used for general comparisons between the number of levels that should be used in the underlying hierarchy.

Figure 3 shows similar experiments but with 100,000 total clients (20,000 of them active) rather than 1,000,000. Comparing these results to those with more clients provides intuition about the effects of scaling the client population, which may help predict system behavior for populations larger than the 1,000,000 that we are able to simulate.

- As expected, increasing (decreasing) the total number of clients decreases (increases) the per-client request rate for which hierarchies begin to pay off rel-

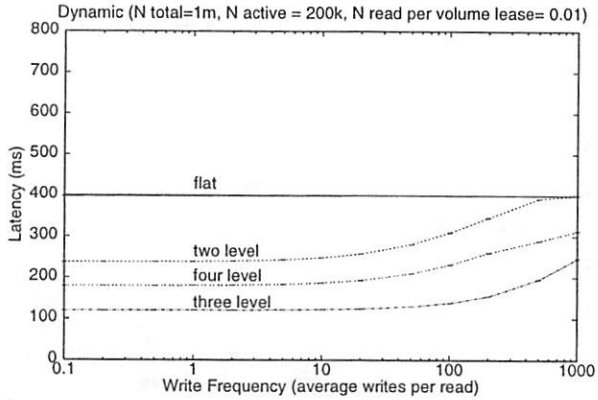


Figure 5: Average read latency under the dynamic algorithm as the write frequency for 10 objects in a server volume is varied. Average read frequency per client is fixed at 0.01 read per client per volume lease period.

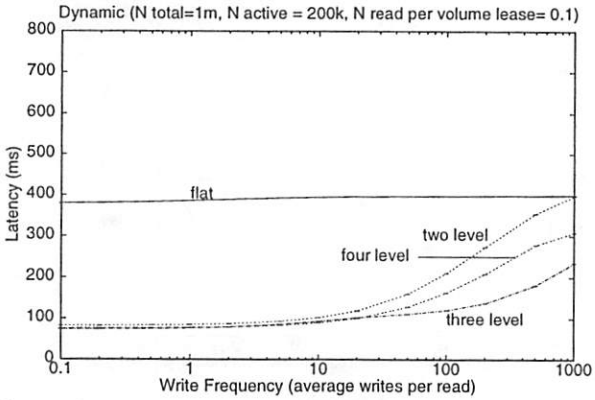


Figure 6: Same as Figure 5, except that the read frequency is 0.1 read per client per volume lease period.

ative to the flat Volume Leases configuration. We observe similar results when we vary the number of active clients (graph omitted).

Figure 4 shows how server load varies with client request rate hierarchies spanning one million clients. (Results for varying the number of active clients or simulating a universe of 100,000 clients are omitted, but are qualitatively similar). Assuming that a server can handle a few thousand requests per volume lease period, we conclude:

- The flat Volume Leases algorithm scales to tens of thousands of clients under workloads corresponding to a range of reasonable web access patterns.
- The addition of hierarchies supports scalability to many millions of clients under nearly arbitrary workloads because it bounds the rate of requests at the root to one request per volume lease period per immediate child of the root.

Writes to multiple-object volumes To make simulating a large scale hierarchy feasible, we have so far con-

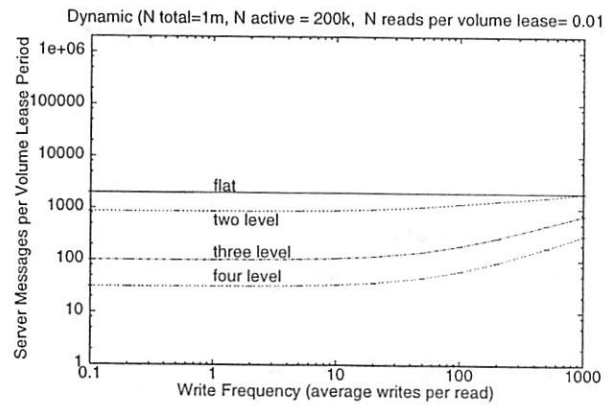


Figure 7: Average server messages per volume lease period under the dynamic algorithm as the write frequency for 10 objects in a server volume is varied. Average read frequency per client is fixed at 0.01 read per client per volume lease period.

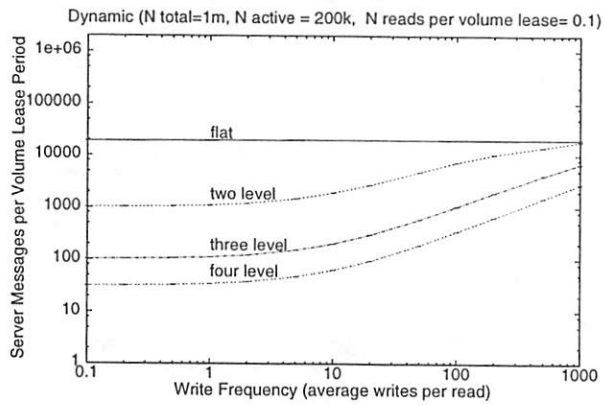


Figure 8: Same as Figure 7, except that the read frequency per client is 0.1 read per client per volume lease period.

sidered only the cost of renewing volume leases. We have not examined the cost of renewing or invalidating object leases. This simplification affects our results in two ways. First, it causes us to understate average read time because in reality clients will sometimes have valid volume leases but still need to fetch object leases. This effect should be modest because object leases are much longer than volume leases. Second, this simplification may cause us to understate the benefit of consistency proxies, particularly when read frequency is low, because consistency proxies can cache long object leases more effectively than short volume leases. To calibrate the effect of object leases for popular servers, we run several simulations with multiple object leases per volume. The results are illustrated in Figures 5 to 10. Due to space constraints, we show the graphs for the dynamic hierarchy algorithm and omit those for the static algorithm; the static results differ little from the dynamic ones.

In these experiments, the server volume contains 10 objects. Each object is modified independently with average write frequency varying from 0.1 writes to 1000 writes per client read. We illustrate performance for

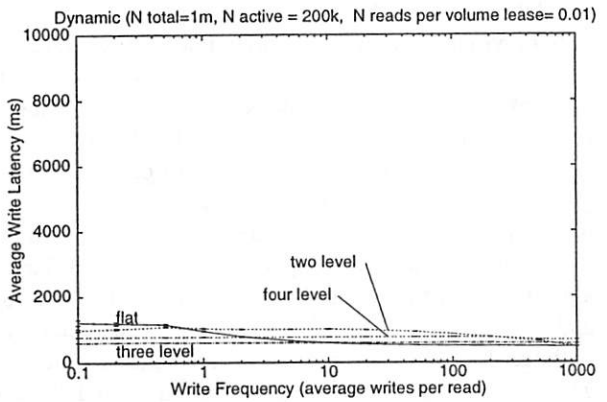


Figure 9: Average write latency seen by server for the dynamic algorithm as the write frequency for 10 objects in a server volume is varied. Average read frequency per client is fixed at 0.01 read per client per volume lease period.

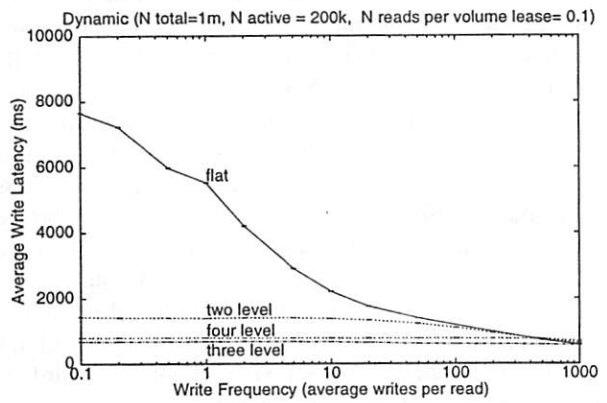


Figure 10: Same as Figure 9, except that the read frequency per client is 0.1 read per client per volume lease period.

servers that have average 0.01 read and 0.1 read per volume lease per client. For example, a system with 30 second volume leases, 3000 seconds between client reads, and 300 seconds between writes would correspond to the *Write Frequency = 10* point in Figure 5.

Figure 5 shows the average read latency as the write frequency changes. Each client issues an average of 0.01 reads per volume lease period. When write frequency is between 0.1 to 10 writes per client read, the results closely match our simplified experiment. Only after write frequency gets higher than 10 writes per client read does the read latency increase become noticeable. Figure 6 is similar to Figure 5, except the read frequency is set to 0.1 reads per volume lease period instead of 0.01. Figures 7 and 8 show the average server load under the same workload as Figures 5 and 6.

Figures 9 and 10 show the average write latency as the write frequency changes. We calculate the write latency by finding the critical path from when the server sends its first invalidation until it receives the final reply. At each node, N, of the hierarchy, we charge $writeCost(N) = nValidChildren(N) \cdot costPerChild + latencyToChild(N) + \max_{\forall c | c \in children(N)}(writeCost(c))$ where $nValidChildren(N)$ is the number of N's direct children that have valid volume leases. Note that by using the *delayed invalidation* optimization [20], the sending of invalidation messages to nodes whose volume leases have expired can be removed from the critical path. The latency to a child is determined by the topology according to our standard formula, and the cost per child is set to 1 ms.

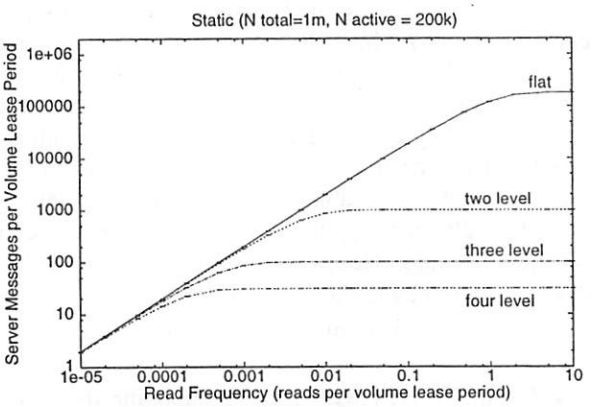


Figure 11: Server lease renewal load as the per-client read frequency is varied for a static server cluster hierarchy serving one million clients, of which 200,000 access the volume in question.

The data indicate three main effects:

- When write frequency increases, the benefit of hierarchy for reads is eroded.
- As write frequency decreases, the write latency increases. This is because the number of valid leases that accumulate between writes, and thus must be invalidated, increases.
- For a frequently accessed popular server, flat volume leases can introduce significant write delays, but hierarchies can remedy this problem.

4.2 Server cluster

The hierarchical consistency mechanisms can be used not only to distribute consistency algorithms across a WAN, but also to split a consistency service across a clustered web server on a local area network (LAN) or system area network (SAN). Although an algorithm optimized for splitting consistency state and load across a cluster with a fast network might marginally outperform our more general mechanisms, such an algorithm must solve the same basic problems of fault tolerance, distributing invalidations, gathering acknowledgments, and partitioning state that our algorithm handles, so the simplicity of using a single framework for both LAN and WAN distribution appears attractive.

Figure 11 shows the load on the server in the server cluster hierarchy where the server and all of the internal nodes of the consistency hierarchy are located in a tightly-coupled cluster, and the lowest internal nodes in the hierarchy communicate across a WAN with the clients. This configuration is not designed to improve latency, just load-scalability. As a result, read latency cannot be affected significantly by modifying the consistency structure. Hence, it is not necessary to build a dynamic hierarchy in this circumstance.

Based on this experiment, we conclude:

- For the server-cluster configuration, the static hierarchy (with split and join for fault tolerance) provides a simple mechanism to scale the flat volume leases algorithm by distributing it across a group of nodes in a cluster; dynamic configuration to minimize latency is not required.

4.3 Server-proxy-client

Figures 12 and 13 show the latency and load measurements when the hierarchy algorithms are run on the server-proxy-client underlying hierarchy with one million clients grouped into 100 proxy-groups of 10,000 clients per group. This experiment suggests two points:

- For low read frequencies, the dynamic hierarchy where clients may contact servers directly has a modest advantage over the static hierarchy.

- At high read frequencies both the static and dynamic configurations significantly outperform the flat configuration.

Figure 14 shows latency for several selected volumes under a trace workload. The workload is the DEC trace [4], and we configure the system with all clients under a single proxy. We map each server in the trace to a different volume. We present results for 8 selected servers: the 4 most popular ones and 4 of medium popularity.

- The trace workloads include multiple objects per volume, and long object leases are easier to cache in a hierarchy. As a result, the static hierarchy begins to pay dividends even with relatively low access rates.

This suggests that for many current web workloads, the simple static hierarchy using the simple server-proxy-client hierarchy may be a reasonable deployment option. This configuration might also provide a practical way to traverse firewalls to deliver consistency signals.

5 Related work

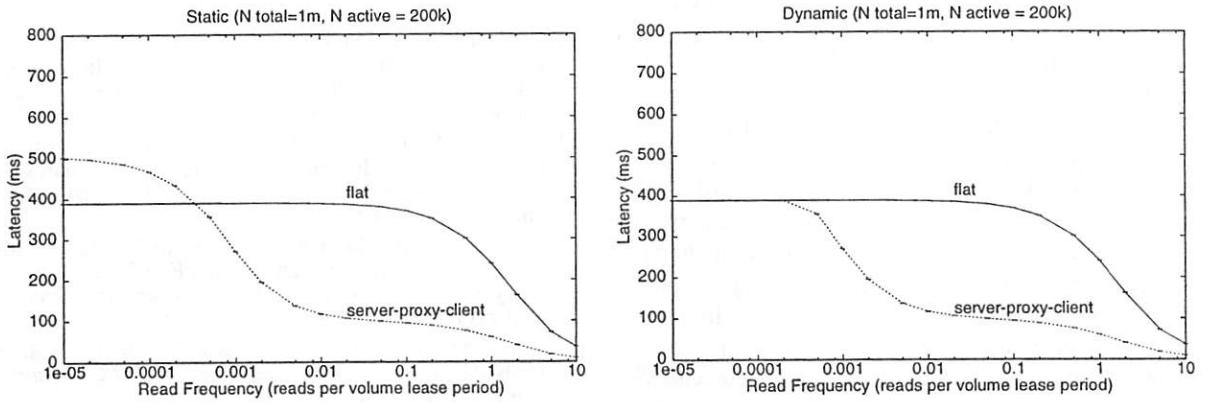
In previous work, we compared non-hierarchical consistency algorithms based on volume leases to traditional callback and polling algorithms. We found that algorithms based on volume leases could both (i) significantly outperform traditional callback or object lease algorithms for a given maximum tolerable server write delay; and (ii) could provide stronger consistency guarantees with performance competitive with callback-based algorithms. Earlier studies by Gwertzman and Seltzer [8] and by Liu and Cao [13] also compare callbacks to polling. Liu and Cao find the performance of the two approaches to be competitive. Gwertzman and Seltzer find that polling with adaptive timeouts can outperform callbacks, but that to gain this advantage the polling algorithm may return stale data more than 4% of the time.

Worrell [19] compares callback and polling protocols in a hierarchical caching system and concludes that the callback algorithms have performance competitive with polling for reasonable time-out values.

Yu et. al [21] independently develop a consistency scheme based on hierarchy, leases, and volumes; this proposal shares many properties with ours, but it differs from ours in three main ways. First, it places a bound on object staleness, whereas our algorithm can be used either to provide strong consistency or to bound staleness [20]. Second, its reconnection protocol requires a client that becomes disconnected to discard all volume and object leases and renew them individually. Third, its consistency servers periodically multicast volume lease renewals and recent object invalidations to their children. Compared to client-initiated volume lease renewal, their approach "pushes" renewals to clients that are not currently accessing a volume; it may thus improve read latency while increasing network traffic and client overheads. Beyond these algorithmic differences, the experimental focus of the study complements ours. Yu et. al primarily focus on comparing the performance of hierarchical invalidation to polling, whereas we focus on understanding the scalability properties of hierarchies.

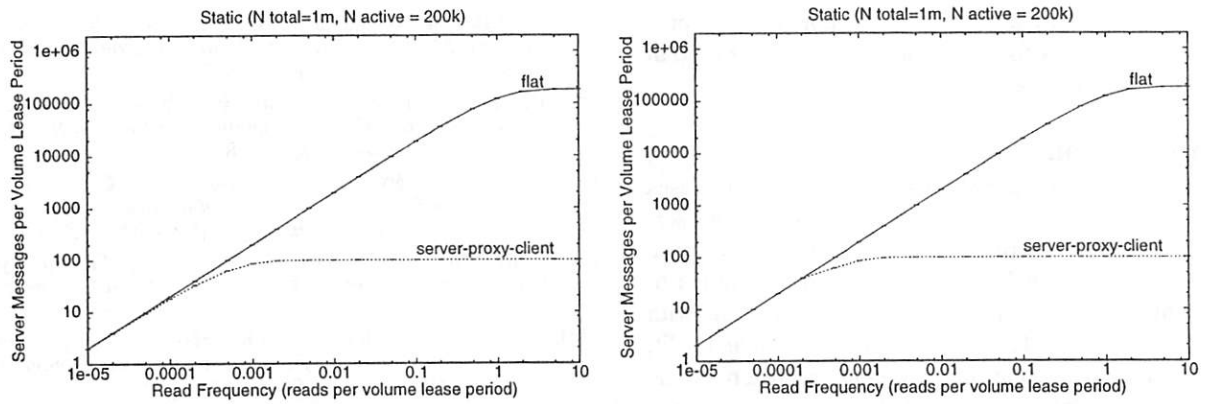
Cohen et. al [3] study the use of volumes for prefetching and consistency. The consistency algorithms they examine are best-effort algorithms based on client polling. Some of their prefetching techniques might also be useful for "prefetching" volume lease renewals in our system. We speculate that adding such prefetching to our system would reduce the read latency cost of hierarchies but magnify the value of hierarchies in reducing server load. Exploring this combination appears to be an interesting area for further study.

Krishnamurthy and Wills [12] examine ways to improve polling-based consistency by piggy-backing optional invalidation messages on other traffic between a client and server. Our volume-based approach allows *de-*



(a) Static (b) Dynamic

Figure 12: Average read latency as the per-client read frequency is varied for a server-proxy-client hierarchy of one million clients, of which 200,000 access the volume in question. In the server-proxy-client hierarchy the internal nodes in the consistency hierarchy are all proxies serving 10,000 clients each.



(a) Static (b) Dynamic

Figure 13: Server lease renewal load as the per-client read frequency is varied for a server-proxy-client hierarchy serving one million clients, of which 200,000 access the volume in question.

	Med 1			Med 2			Med 3			Med 4		
	flat	static	dyn	flat	static	dyn	flat	static	dyn	flat	static	dyn
Latency (ms)	160.5	129.4	135.4	99.0	89.5	92.1	55.6	61.2	57.3	276.3	297.0	279.7
Load (server msgs/read)	0.41	0.23	0.27	0.25	0.16	0.20	0.14	0.12	0.14	0.69	0.57	0.64

(a) Trace results for four medium-loaded volumes.

	Large 1			Large 2			Large 3			Large 4		
	flat	static	dyn	flat	static	dyn	flat	static	dyn	flat	static	dyn
Latency (ms)	84.1	30.8	30.7	123.2	51.1	51.2	133.0	46.7	46.7	68.9	39.3	39.6
Load (server msgs/read)	0.21	0.03	0.03	0.31	0.05	0.05	0.33	0.03	0.03	0.18	0.06	0.07

(b) Trace results for four heavily-loaded volumes.

Figure 14: Average read latency and fraction of renewal requests sent to the server for the four medium-loaded and four heavily-loaded volumes from the DEC trace workload under a server/proxy/client hierarchy in which the internal node in the consistency hierarchy is the proxy serving the DEC clients.

layed invalidations [20] where servers delay object invalidation messages to clients whose volume leases have expired. Combining delayed invalidations with piggybacking may be another useful optimization.

Cache consistency protocols have long been studied for distributed file systems [9, 15, 17]. Several aspects of Coda's [11] consistency protocol are reflected in our algorithms. In particular, our notion of a volume is similar to that used in Coda [14]. However, ours differ in two key respects. First, Coda does not associate volumes with leases, and relies instead on other methods to determine when servers and clients become disconnected. Second, because Coda is designed for different workloads, its design trade-offs are different. For example, because Coda expects clients to communicate with a small number of servers and it regards disconnection as a common occurrence, Coda aggressively attempts to set up volume callbacks to all servers on each hoard walk.

Our reconnection protocol in which clients help servers recover the state they need is based on the server crash recovery protocol in Sprite [1].

Finally, we note that Volume Leases on the set of all objects provided by a server can be thought of as providing a framework for the "heartbeat" messages used in many distributed state systems.

6 Conclusions

In this paper we have shown that the Volume Leases algorithm can provide strong consistency for Internet services with hundreds of thousands of clients. We have also shown how the Volume Leases can be applied to hierarchical caches to perform well for workloads with millions of clients. The key mechanisms, join and split, can be implemented using a simple extension of the Volume Leases algorithm. Finally, we have evaluated a number of hierarchy configurations, and our results show that a dynamically configurable hierarchy provides tremendous amounts of scalability.

Acknowledgements

We thank the anonymous USITS reviewers and our shepherd, Peter Honeyman, for their valuable feedback on earlier drafts of this work.

References

[1] M. Baker. *Fast Crash Recovery in Distributed File Systems*. PhD thesis, University of California at Berkeley, 1994.

[2] A. Chankhunthod, P. Danzig, C. Neerdaels, M. Schwartz, and K. Worrell. A Hierarchical Internet Object Cache. In *Proc. of the 1996 USENIX Technical Conf.*, January 1996.

[3] E. Cohen, B. Krishnamurthy, and J. Rexford. Improving End-to-End Performance of the Web Using Server Volumes and Proxy Filters. In *Proc. of the ACM SIGCOMM '98 Conf. on Applications, Technologies, Architectures, and Protocols for Computer Communication*, 1998.

[4] Digital Equipment Corporation. Digital's Web Proxy Traces. `ftp://ftp.digital.com/pub/DEC/traces/proxy/webtraces.html`, September 1996.

[5] B. Duska, D. Marwood, and M. Feeley. The Measured Access Characteristics of World-Wide-Web Client Proxy Caches. In *Proc. of the USENIX Symposium on Internet Technologies and Systems*, December 1997.

[6] S. Gadde, J. Chase, and M. Rabinovich. Directory Structures for Scalable Internet Caches. Technical Report CS-1997-18, Duke University Department of Computer Science, November 1997.

[7] C. Gray and D. Cheriton. Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency. In *Proc. of the 12th ACM Symposium on Operating Systems Principles*, pages 202–210, 1989.

[8] J. Gwertzman and M. Seltzer. World-Wide Web Cache Consistency. In *Proc. of the 1996 USENIX Technical Conf.*, January 1996.

[9] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and Performance in a Distributed File System. *ACM Trans. on Computer Systems*, 6(1):51–81, February 1988.

[10] Kermarrec, Kuz, van Steen, and Tanenbaum. A Framework for Consistent, Replicated Web Objects. In *Proc. of the 18th Intl. Conf. on Distributed Computing Systems*, 1998.

[11] J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. *ACM Trans. on Computer Systems*, 10(1):3–25, February 1992.

[12] B. Krishnamurthy and C. Wills. Piggyback Server Invalidation for Proxy Cache Coherency. In *Proc. of the 7th Intl. World Wide Web Conf.*, 1998.

[13] C. Liu and P. Cao. Maintaining Strong Cache Consistency in the World-Wide Web. In *Proc. of the Seventeenth Intl. Conf. on Distributed Computing Systems*, May 1997.

[14] L. Mummert and M. Satyanarayanan. Large Granularity Cache Coherence for Intermittent Connectivity. In *Proc. of the Summer 1994 USENIX Conf.*, June 1994.

[15] M. Nelson, B. Welch, and J. Ousterhout. Caching in the Sprite Network File System. *ACM Trans. on Computer Systems*, 6(1), February 1988.

[16] C. Plaxton, R. Rajaram, and A. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *Proc. of the 9th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 311–320, June 1997.

[17] V. Srinivasan and J. Mogul. Spritely NFS: Experiments with Cache Consistency Protocols. In *Proc. of the 12th ACM Symposium on Operating Systems Principles*, pages 45–57, December 1989.

[18] R. Tewari, M. Dahlin, H. Vin, and J. Kay. Design Considerations for Distributed Caching on the Internet. In *Proc. of the 19th Intl. Conf. on Distributed Computing Systems*, May 1999.

[19] K. Worrell. Invalidation in Large Scale Network Object Caches. Master's thesis, University of Colorado, Boulder, 1994.

[20] J. Yin, L. Alvisi, M. Dahlin, and C. Lin. Using Leases to Support Server-Driven Consistency in Large-Scale Systems. In *Proc. of the 18th Intl. Conf. on Distributed Computing Systems*, May 1998.

[21] H. Yu, L. Breslau, and S. Schenker. A Scalable Web Cache Consistency Architecture. In *Proc. of the ACM SIGCOMM '98 Conf. on Applications, Technologies, Architectures, and Protocols for Computer Communication*, September 1999.

Organization-Based Analysis of Web-Object Sharing and Caching

Alec Wolman, Geoff Voelker, Nitin Sharma, Neal Cardwell, Molly Brown,
Tashana Landray, Denise Pinnel, Anna Karlin, Henry Levy

Department of Computer Science and Engineering
University of Washington

Abstract

Performance-enhancing mechanisms in the World Wide Web primarily exploit repeated requests to Web documents by multiple clients. However, little is known about patterns of shared document access, particularly from diverse client populations. The principal goal of this paper is to examine the sharing of Web documents from an *organizational* point of view. An organizational analysis of sharing is important, because caching is often performed on an organizational basis; i.e., proxies are typically placed in front of large and small companies, universities, departments, and so on. Unfortunately, simultaneous multi-organizational traces do not currently exist and are difficult to obtain in practice.

The goal of this paper is to explore the extent of document sharing (1) among clients within single organizations, and (2) among clients across different organizations. To perform the study, we use a large university as a model of a diverse collection of organizations. Within our university, we have traced all external Web requests and responses, anonymizing the data but preserving organizational membership information. This permits us to analyze both inter- and intra-organization document sharing and to test whether organization membership is significant. As well, we characterize a number of parameters of our data, including basic object characteristics, object cacheability, and server distributions.

1 Introduction

The need to understand Web behavior and performance has led to a large number of studies, aimed in particular at classifying Web document *characteristics* [11, 12, 13, 16, 21]. In contrast, the principal goal of this study is to evaluate document *sharing behavior* on the Web, both *within* organizations and *between* organizations. By document sharing, we mean access to the same Web documents by different clients. Sharing behavior has obvious implications for performance, particularly with respect to the effectiveness of proxy caching (e.g., [9, 14, 17, 20, 27]). Proxy caches are often located at organizational boundaries and improve performance only if many documents are shared by many clients. Therefore, an understanding of sharing gives us added insight into potential performance-enhancing mechanisms and alternative caching structures.

An analysis of document sharing within an organization is straightforward and can help predict the benefits of an organizational proxy cache [13]. Studying sharing across multiple organizations is much more difficult, however. Tracing of the entire Web is obviously not achievable, but even simultaneous traces of multiple organizations do not currently exist. In addition, the requirement of most organizations for anonymization of URLs and IP addresses, along with the different dates of data capture, makes correlation of separate traces challenging, if not impossible.

In this study, we use The University of Washington (UW) as a basis for modeling intra- and inter-organizational Web-object sharing. The UW is the largest university in the northwest part of the U.S., with a population of over 50,000 people, including 35,000 students, 10,000 full-time staff, and 5,000 faculty. The university has a large communications infrastructure, consisting of thousands of computers connected via both high-speed networks and modems. Together, this community generates a workload of about 17,400 university-external Web requests per minute at peak periods.

As with other universities, UW is organized into many colleges, departments, and programs, each with its own disparate administrative, academic, or research focus. For example, the UW includes museums of art and natural history, medical and dental schools, libraries, administrative organizations, and of course academic departments, such as music, Scandinavian languages, and computer science. What do such diverse organizations have in common with respect to their Web access requests? To answer this question, we have traced all UW-external Web requests; we anonymize the data in such a way as to identify requests (and associated responses) with the 170 or so independent organizations from which they were issued. This permits us to study organization-specific document access and sharing behavior. We have collected a number of traces during the period from October 1998 through the present. In general, all of our traces show the same basic patterns. The results in this paper are based on a representative one-week trace taken in mid-May 1999, and therefore show the very latest character-

istics of modern Web traffic.

The paper is organized as follows. The next section provides a brief description of related work. In Section 3 we describe our trace-capture methodology. Section 4 contains a high-level description of the workload we traced. Section 5 focuses on organization-based statistics and also provides inter- and intra-organization sharing analysis. In Section 6 we discuss cacheability of documents, and reasons why documents are not cacheable. Finally, Section 7 summarizes our study and its results.

2 Previous Work

Numerous recent studies of Web traffic have been performed. These studies include analyses of Web access traces from the perspective of browsers [11, 21], proxies [2, 4, 6, 10, 12, 15, 18, 19, 24], and servers [1, 3, 23]. The earlier tracing studies were rather limited in request rate, number of requests, and diversity of population. The most recent tracing studies have been larger and generally more diverse. In addition to static analysis, some studies have also used trace-driven cache simulation to characterize the locality and sharing properties of these very large traces [2, 5, 13, 15, 16, 19], and to study the effects of cookies, aborted connections, and persistent connections on the performance of proxy caching [5, 15].

In this paper, we expand on these previous research efforts. Our focus is on sharing and cacheability; however we can also compare our current HTTP traffic characteristics to earlier studies, showing how the Web workload has changed. Our work is based on the most recent data from a large diverse population. More important, we preserve enough information so that we can analyze requests with respect to inter-organization and intra-organization document sharing.

3 Measurement Methodology

We use *passive network monitoring* to collect our traces of Web traffic traveling between the University of Washington and the rest of the Internet. UW connects to its Internet Service Providers via two border routers; one router handles primarily outbound traffic and the other inbound traffic. These two routers are fully connected to four 100-megabit Ethernet switches distributed across the campus. Each switch has a monitoring port that is used to send copies of the incoming and outgoing packets to our monitoring host, which analyzes the packets and produces a trace.

We designed and implemented the tracing software used to produce that data in this study. Our user-level tracing software runs on a 500 MHz Digital Alpha 21164 workstation running Digital Unix V4.0. This software installs a kernel packet filter [22] to deliver all TCP packets from the network interfaces to the user-level monitoring process, which analyzes the packets and produces a trace. The user-level process consists of three layers: TCP segment analysis, HTTP header processing, and logging. The TCP segment analysis layer classifies individual TCP packets into TCP connections and identifies the first data segments in each connection. The first data segment is used to decide whether or not the connection is an HTTP connection. This technique allows us to see all HTTP traffic (not just port 80). Once a connection has been classified as an HTTP connection, we monitor further segments on that connection so that we can locate all the relevant HTTP headers when persistent connections are in use. The HTTP header processing layer is responsible for parsing the HTTP headers extracted from TCP data segments in the HTTP connection. Once the headers have been parsed, we extract the fields to be saved and anonymize those fields that contain sensitive information. We also anonymize the IP addresses here, and then pass that information to the logging layer. The logging layer takes the information from the HTTP parser, converts it to a compact binary representation, compresses it, and writes it to disk. We maintain packet loss counters on the monitoring host at the device driver level, at the packet filter level, and at user level. During the May trace, we measured the packet loss at .0007%. It is also possible for the switches to drop packets, and we cannot detect packet loss at the these switches, but the UW network administrators who manage the switches tell us that they have significant excess capacity.

We use an anonymization approach that protects privacy but preserves some address locality information. For internal addresses, we classify the IP address based on its "organization" membership. An organization is a set of university IP addresses that forms an administrative entity; an organization may include multiple subnets. For instance, all machines in the Computer Science Department are in a single organization, machines in the Department of Dentistry are in another, and machines connected to the campus Museum of Natural History are in yet another. We constructed the mapping from subnets to organization identifiers based on information obtained from the campus network administrators. Once the organization identifiers are assigned, both the IP address and the organization identifier are anonymized. Furthermore, some bits of information in the IP address are destroyed before anonymization to make the anonymization more secure. If the hash function or key is compromised, no transaction can be associated with a client address with absolute certainty.

For external addresses, we anonymize each octet of the server IP address separately. For our purposes, two servers are near each other if they share most or all of the Internet path between them and the university. We con-

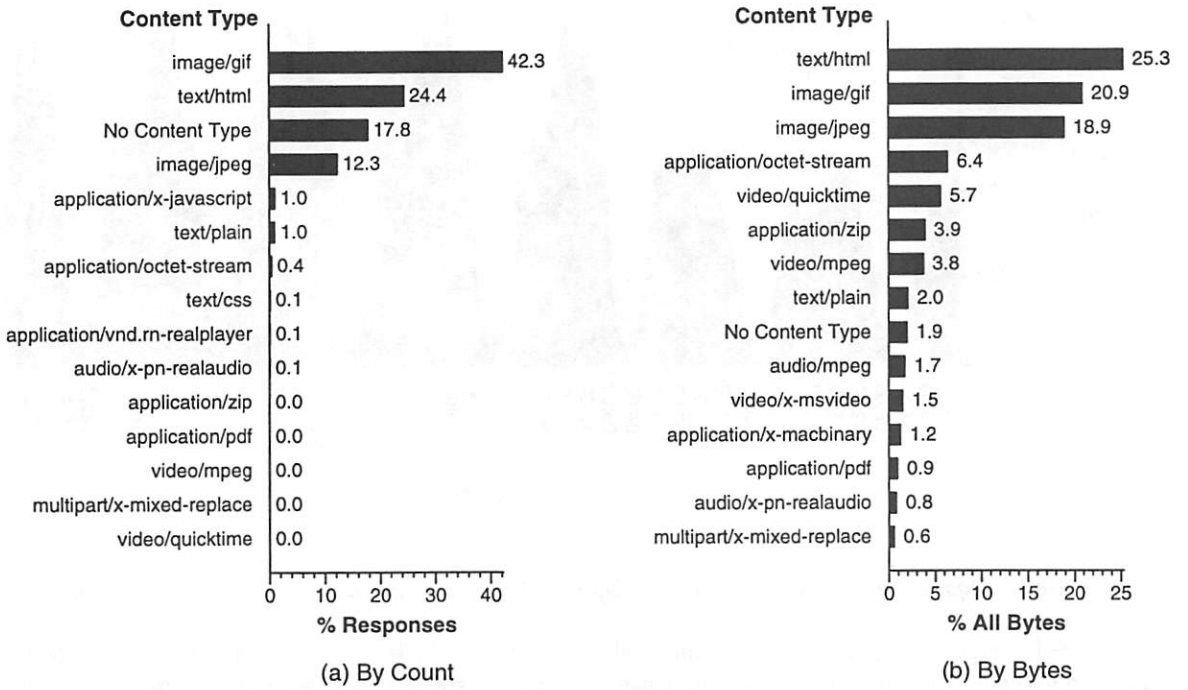


Figure 1: Histogram of the top 15 content types by count and size.

sider two servers to be on the same subnet when the first three octets of their IP addresses match. Given the use of classless routing in the Internet, this scheme will not provide 100% accuracy, but for large organizations we expect that this assumption will be overly conservative rather than overly aggressive.

Although our tracing software records all HTTP requests and responses flowing both in and out of UW, the data presented in this paper is filtered to only look at HTTP requests generated by clients inside UW, and the corresponding HTTP responses generated by servers outside of UW. All of our results are based on the entire trace collected from Friday May 7th through Friday May 14th, 1999, except for the organization-based sharing results in Section 5, which are from a single day (Tuesday) of our trace (the limitation is due to the memory requirements of the sharing analysis).

4 High-Level Data Characteristics

Table 1 shows the basic data characteristics. As the table shows, our trace software saw the transfer of 677 gigabytes of data in response packets, requested from about 23,000 client addresses, and returned from 244,000 servers. It is interesting that, compared to the commonly-used 1996 DEC trace (analyzed, e.g., in [13]), which had a similar client population, we saw four times as many requests in one week as DEC saw in 3 weeks. These requests and corresponding response and close events follow the typical diurnal cycle, with a minimum of 460 requests per minute (at 5 AM) and a peak of 17,400 requests per minute (at 3 PM).

Figures 1a and 1b present a histogram of the top content types by object count and bytes transmitted, respectively. By count, the top four are image/gif, text/html, No Content Type, and image/jpeg, with all the rest of the content types at significantly lower numbers. The No Content Type traffic, which accounts for 18% of the responses, consists primarily of short control messages. The largest percentage of bytes transferred is accounted for by text/html with 25%, though the sum of the image/gif (19%) and image/jpeg (21%) types accounts for 40% of the bytes transferred. The remaining content types account for decreasing numbers of bytes with a heavy-tailed distribution.

Another type that accounts for significant traffic, which is not readily apparent from the table, is multimedia content (audio and video). The sum of all 59 different audio and video content types that we observed during the May trace adds up to 14% of all bytes transferred. In

HTTP Transactions (Requests)	82.8 million
Objects	18.4 million
Clients	22,984
Servers	244,211
Total Bytes	677 GB
Average requests/minute	8,200
Peak requests/minute	17,400

Table 1: Overall statistics for the one-week trace.

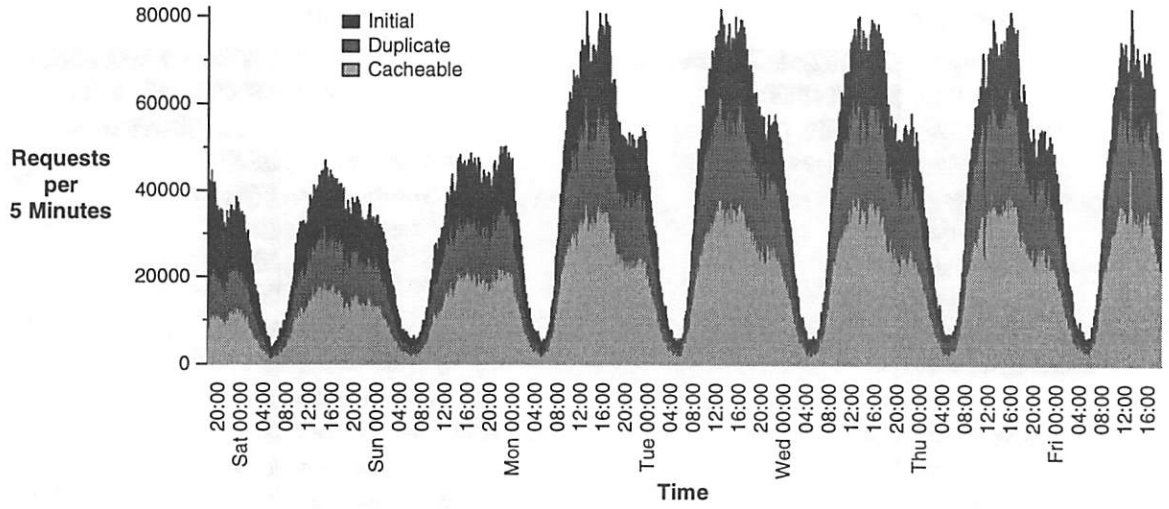


Figure 2: Requests broken down into initial, duplicate, and cacheable duplicate requests over time.

addition, there is a significant amount of streaming multimedia content that is delivered through an out-of-band channel between the audio/video player and the server.

In a preliminary attempt to view some of this out-of-band multimedia traffic, we extended our tracing software to analyze connections made by the Real Networks audio/video player, examining port 7070 traffic. Newer versions of the Real Networks player use the RTSP protocol, which we do not handle. The Real Networks player sets up a TCP control connection on port 7070, and then transfers the data on UDP port 7070. Our trace software only collects TCP segments, so we analyze the control connection to determine how much data is being transferred. When the control connection is shut down, a "statistics" packet is transmitted that contains the average bandwidth delivered (in bits per second) as measured by the client for the completed connection. We take that that bit-rate and multiply it by the connection duration time to estimate the size of the content transferred. Some of the control connections do not transmit the statistics packet, in which case we cannot make an estimate.

During the week of the May trace, we observed 55000 connections, of which approximately 40% had statistics packets. For those 40%, we calculated that 28 GB of Real-Audio and Real-Video data were transferred (which would scale to 10% of the amount of HTTP data transferred if the other 60% of connections have similar characteristics). Furthermore, the Real-Audio and Real-Video objects themselves are quite large, with an average size of just under a megabyte. When we sum up all the different kinds of multimedia content, we see that from 18% to 24% of Web related traffic coming in to the University is multimedia content, and this is a lower bound since we know that we're missing RTSP traffic at the very least. We believe that the large quantity of audio and video is signaling a new trend; e.g., in the data collected for studies reported in [12] and [16], audio traffic does not appear.

We were also curious about the HTTP protocol versions currently in use. The majority of requests in our trace (53%) are made using HTTP 1.0, but the majority of responses use HTTP 1.1 (69%). In terms of bytes transferred, the majority of bytes (75%) are returned from HTTP 1.1 servers.

These statistics simply serve to provide some background about the general nature of the trace data, in order to set the context for the analysis in the next two sections.

5 Analysis of Document Sharing

This section presents and analyzes our trace data, focusing on document sharing. As previously stated, our intention is to use the university organizations as a simple model of independent organizations in the Internet. Our goal is to answer several key questions with respect to Web-document sharing, for example:

1. How much object sharing occurs between different organizations?
2. What types of objects are shared?
3. How are objects shared in time?
4. Is membership in an organization a predictor of sharing behavior?
5. Are members of organizations more similar to each other than to members of different organizations, or do all clients behave more-or-less identically in their request behavior?

Figure 2 plots total Web requests per 5 minute period over the one-week trace period. The shading of the graph

divides the curve into three areas: the darkest portion shows the fraction of requests that are initial (first) requests to objects, while the medium grey portion shows the subset that are duplicate (repeated) requests to documents. A request is considered a duplicate if it is to a document previously requested in the trace by any client. The lightest grey color shows those requests that are both duplicate and cacheable, as we will discuss later.

Overall, the data shows that about 75% of requests are to objects previously requested in the trace. This matches fairly closely the results of Duska et al. on several large organizational traces [13]. The percentage of shared requests rises very slowly over time, as one might expect. From our one-week trace, we cannot yet see the peak; however, this analysis does not consider document timeouts or replacements, therefore the 75% is optimistic if used as a basis for prediction of cache behavior. Furthermore, we cannot tell from the figure how many of the requests to a shared object were duplicate requests from the same client; overall, we found that about 60% of the requests to shared documents were first requests by a client to those documents; 40% were repeated requests by the same client.

A key component of our data is the encoding of the organization number, which allows us to identify each client as belonging to one of the 170 active university organizations. These organizations include academic and administrative departments and programs, dormitories, and the university-wide modem pool. Figures 3a and 3b show the organization size, the request rate, and number of objects accessed by each organization. There are several very large organizations, with most somewhat smaller. The largest organization has 919 "anonymized" clients, the second largest organization is the modem pool with 759 clients, and the third largest organization has 626 clients.[1] The top 20 organizations all have more than 100 clients, as shown by the labels in Figure 5. Because of the way that client IP addresses are anonymized, we cannot uniquely identify an individual client, i.e., each anonymized client address could correspond to up to 4 separate clients. For this trace the ratio of "real" clients to "anonymized" clients measured by the low levels of our trace software is 1.67; therefore, our 13,701 anonymized clients represent 22,984 true clients.

Using the organization data, we can analyze the amount of object sharing that occurs both within and between organizations.

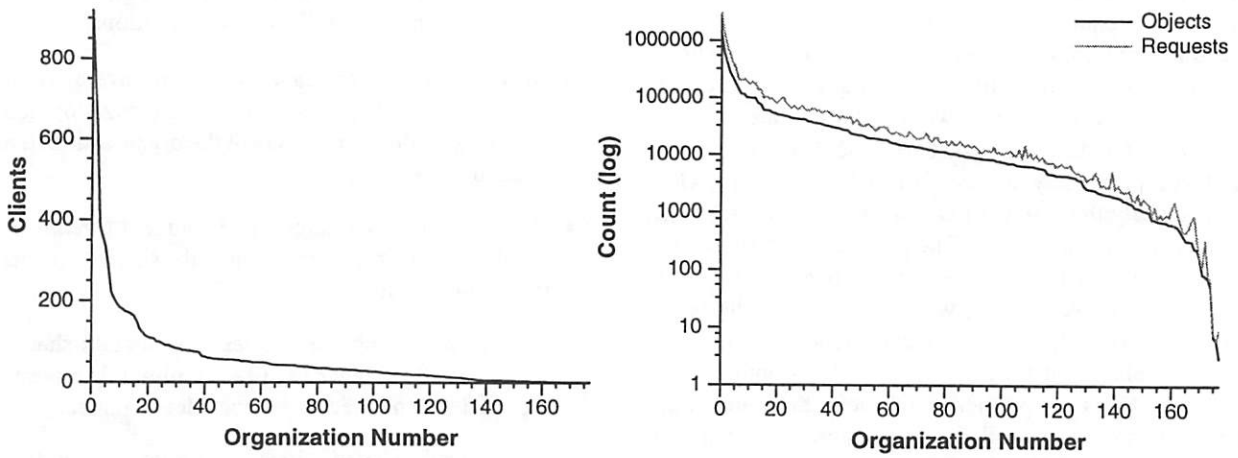
Figure 4a shows intra-organization (*local*) sharing from the perspective of both objects and requests. The black line shows the percentage of all objects accessed by each organization that are *locally-shared* objects, i.e., accessed by more than one organization member. The light grey line shows the percentage of all organization *requests* that are to these locally-shared objects. The organizations are ordered by decreasing locally-shared object percentage. From our data on intra-organization sharing we can make the following observations:

- Only a small percentage (4.8% on average) of the *objects* accessed within an organization are shared by multiple members of the organization (the smooth black line).
- However, a much larger percentage of *requests* (16.4% on average) are to locally-shared objects (the light grey line).
- The average number of requests per locally-shared object is 4.0 – higher than the minimal 2 requests required for an object to be considered shared.
- Each locally-shared object is requested by two clients on average in each organization.

Figure 4b shows the inter-organization (*global*) sharing activity. Here the black line shows the percentage of all objects accessed by each organization that were also accessed by at least one *other* organization; we call such objects *globally-shared* objects. Similarly, the light grey line shows the percentage of all *requests* by an organization to globally-shared objects. The organizations are ordered by decreasing globally-shared object percentage. From our data on inter-organization sharing we can make the following observations:

- There is more sharing with other organizations than within the organization; the fraction of globally-shared objects and requests in Figure 4b is much higher than the locally-shared objects and requests in Figure 4a. This is not surprising, because the combined client population of all of the organizations is significantly larger than any one organization alone. As a result, there is a much greater opportunity for the clients in one organization to share with clients from any of the other organizations.
- For 65% of the organizations, more than half of the objects referenced are globally-shared objects (the smooth black line).
- For 94% of the organizations, more than half of the requests are to globally-shared objects, and for 10% of the organizations 75% of the requests are to globally-shared objects (the light grey line).
- However, globally-shared objects are not requested frequently by each organization. On average, each organization makes 1.5 requests to a globally-shared object.

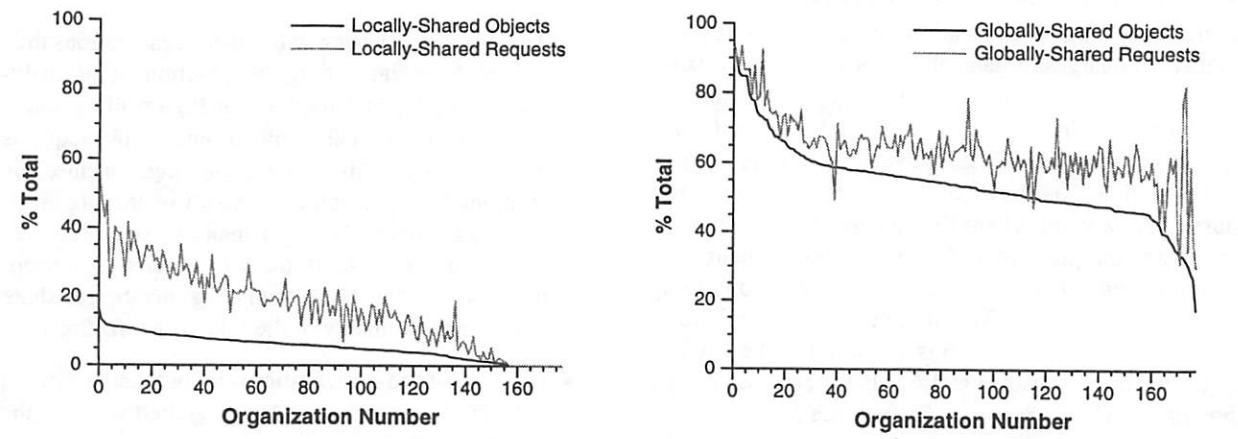
[1] The modem pool is somewhat special, because multiple clients can login through a single IP address in the pool.



(a) Clients Per Organization

(b) Objects and Requests Per Organization

Figure 3: Distribution of clients, objects, and requests in organizations. The object and request graph is sorted by the number of objects in an organization. Note that the y-axis of (b) uses a log scale.



(a) Intra-Organization Sharing (Local)

(b) Inter-Organization Sharing (Global)

Figure 4: The left graph shows the fraction of objects and requests accessed by the organization that are shared by more than one client within the organization. The right graph shows the fraction of objects and requests accessed by the organization that are shared with at least one other organization.

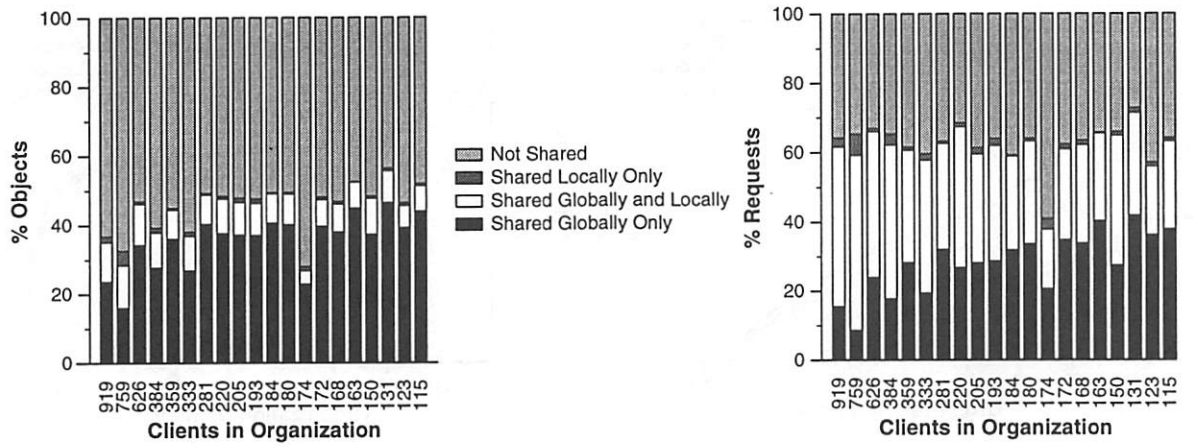


Figure 5: Breakdown of objects (a) and requests (b) into the different categories of sharing, for the 20 largest organizations. The labels on the x-axis show the number of clients in each organization shown.

- On average, a globally-shared object is accessed by only one client in each organization.

A key question raised by these figures is whether the objects shared within an organization are the *same* set of objects that are shared across organizations. Figure 5a shows, for the 20 largest organizations, a breakdown of organization-accessed objects into various sharing categories: local only, global only, local and global, and not shared. Figure 5b shows the same breakdown by request. The graphs are ordered in decreasing organization size, with the organization size shown on the x-axis.

From Figure 5b, we see that the fraction of requests to shared objects is fairly flat across these organization sizes. As we would expect, the fraction that are shared globally-only rises somewhat with decreased organization size, while the fraction that are locally-shared decreases with decreasing organization size. That is, in general, the smaller the organization, the less organization-internal sharing, and the more global sharing. Looking at the white section of the bars in both figures, we see that the small percentage of objects that account for both local and global sharing are very hot, and account for a much greater fraction of the requests than the objects they represent. In contrast, the percentage of requests to objects shared locally-only is very small for these organizations.

To aid in the understanding of the degree of object sharing, Figure 6 plots the number of objects (on the y-axis) that were shared by exactly x organizations. Most objects are accessed by only one organization, as shown by the steepness of the curve at $x = 1$. We also found that there were more than 1000 objects accessed by 20 organizations and more than 100 objects accessed by 45 organizations.

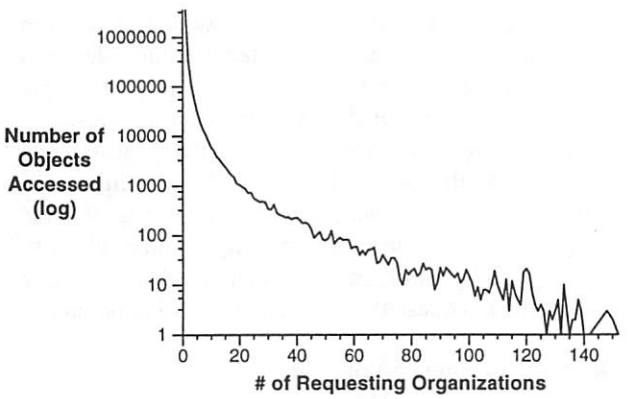
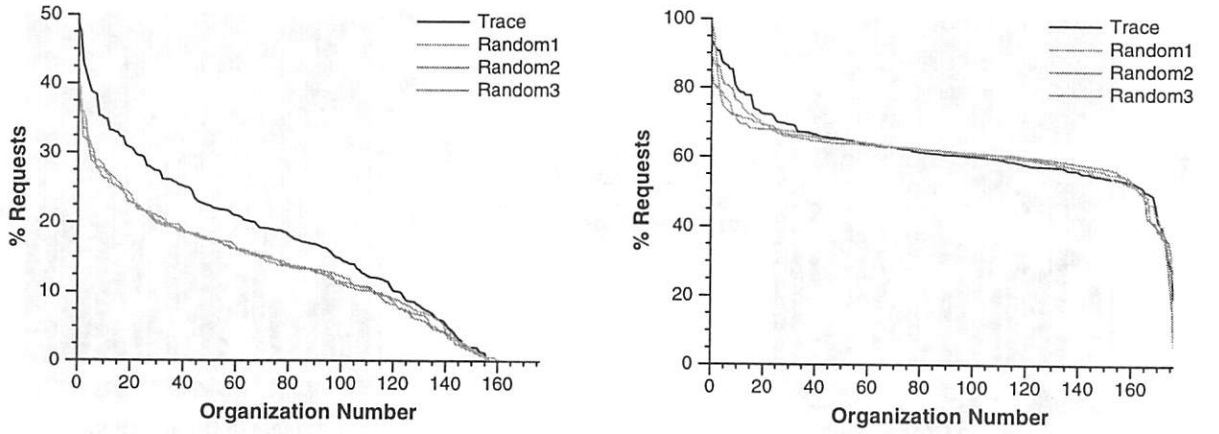


Figure 6: The number of objects accessed by a given number of organizations. Note that the y-axis uses a log scale.

A key question with respect to our sharing data is whether organization membership is significant. To answer this question, we randomly assigned clients to organizations, and compared the inter- and intra-organization sharing in the random assignments with the sharing seen in our trace analysis presented above. (The random organizations had the same sizes as the actual organizations.) Figure 7a plots the fraction of requests to locally-shared objects of the trace organizations and three randomly-assigned organizations. From the figure, we see that sharing is higher in the real organizations than in the randomly-assigned organizations. In other words, there is locality of references in organization membership. Figure 7b plots the fraction of requests to globally-shared objects for the trace and for the three random organizations. As expected, there is no significant difference in the amount of global sharing between the real trace and the randomized organization assignment.



(a) Intra-Organization Sharing (Local)

(b) Inter-Organization Sharing (Global)

Figure 7: Fraction of requests in the organization that are shared within this organization (a) and shared with at least one other organization (b), compared with three random client-to-organization assignments.

The organization-oriented data show that there is, in fact, significance to organization membership. Members of an organization are more likely to request the same documents than a set of clients of the same size chosen at random. However, the vast majority of the requests made are to objects that are *globally* shared. In addition, objects that are shared both locally within an organization and globally with other organizations are more likely to be requested by an organization member. This suggests that the most requested objects are universally popular.

Object and Server Popularity

For another aspect of sharing patterns we examine the servers that are being accessed and server proximity (i.e., which servers are close to each other in the network).

Figure 8 shows the cumulative distribution functions of both server popularity and server subnet popularity, where popularity is measured by the request-count. The byte-count curves for server popularity and server subnet popularity are effectively identical to the request-count curves shown in the graph. The data indicates that 50% of the objects accessed and bytes transferred come from roughly the top 850 servers (out of a total of 244,211 servers accessed). A server subnet is a set of servers that share the same first 24 bits of their IP addresses. Such groups of servers are typically mirrors of each other, or at least sit in a single server farm owned by a single company. We see that 50% of the objects come from about the top 200 server subnets; 18% come from the top 20 subnets.

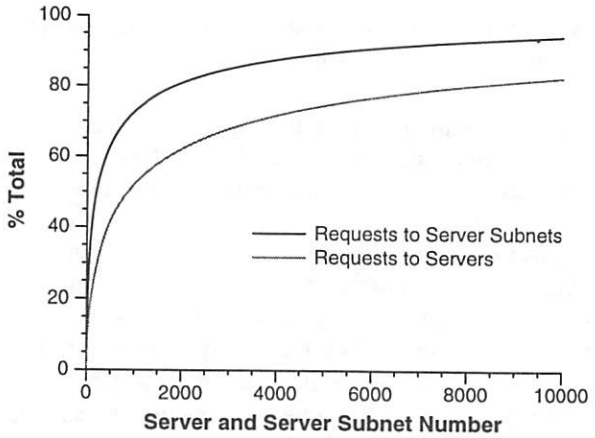


Figure 8: The cumulative distributions of server and server subnet popularity.

6 Document Cacheability

This section examines cacheability of documents, giving us insight into the potential effectiveness of proxy caches in our environment. Web proxy caches are a key performance component of the WWW infrastructure; their objective is to improve performance through caching of documents requested more than once. Proxies typically live at the boundaries of an organization, caching documents for all clients within that organization.

In Figure 2 we saw a time-series graph of the percentage of duplicate requests (i.e., requests to a previously-accessed document) and cacheable requests in our trace. The cacheable requests are those made to documents that would be cached by a standard proxy cache, such as Squid [25]. We found that, in steady state, approximately 45% of the requests are duplicate and cacheable, placing an upper bound on the hit rate. The wide difference between the duplicate line and the cacheable line indicates that only about half of the duplicate requests (which could benefit from caching) are to objects that are cacheable.

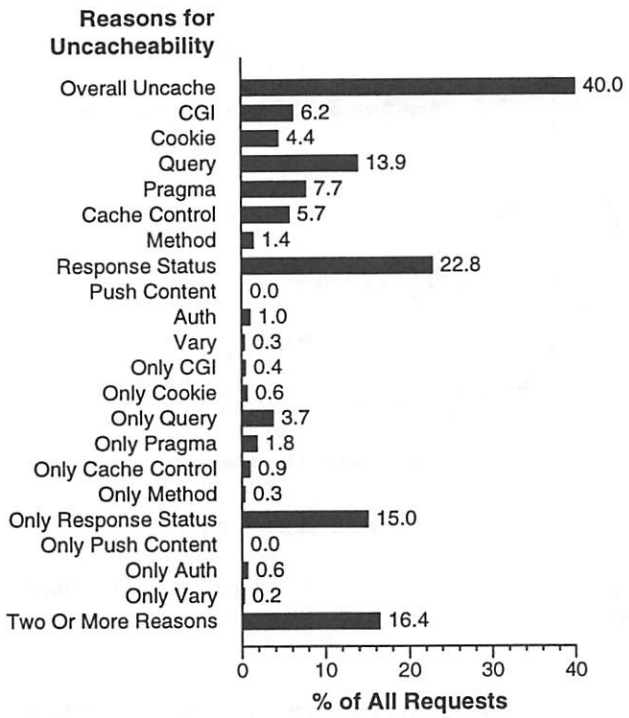


Figure 9: Reasons for uncacheability of HTTP transactions.

Our cacheability analysis is based on the implementation of the Squid proxy cache. We examined the policies implemented by both Squid version 1 and Squid version 2. There are several reasons why a Squid proxy may consider a document uncacheable.

- *CGI* – The document was created by a CGI script or program and is not cached, because it is produced dynamically.
- *Cookie* – The response contains a set-cookie header. Squid version 1 does not allow these responses to be cached, but Squid version 2 does allow them to be cached.
- *Query* – The request is a query, i.e., the object name includes a question mark ("?").
- *Pragma* – The response is explicitly marked uncacheable with a "Pragma: no-cache" header.
- *Cache-Control* – The response is explicitly marked uncacheable with the HTTP 1.1 Cache-Control header.
- *Method* – The request method is not "GET" or "HEAD".
- *Response-Status* – The server response code does not allow the proxy to cache the response. For example, response code 302 (Moved Temporarily) cannot be cached when there is no explicit expiration date specified.

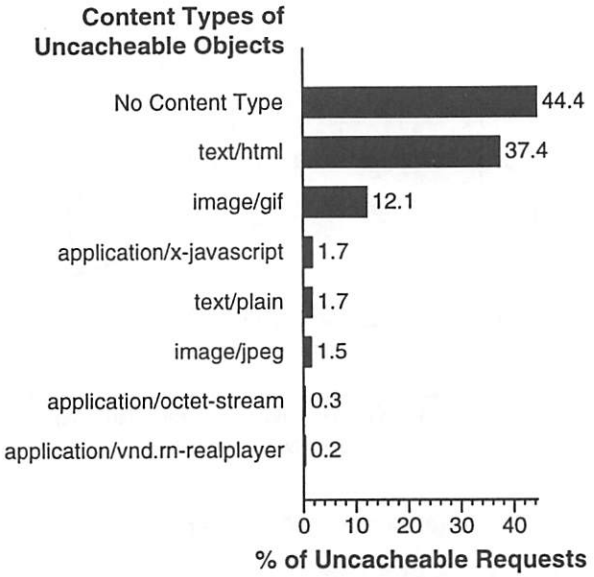
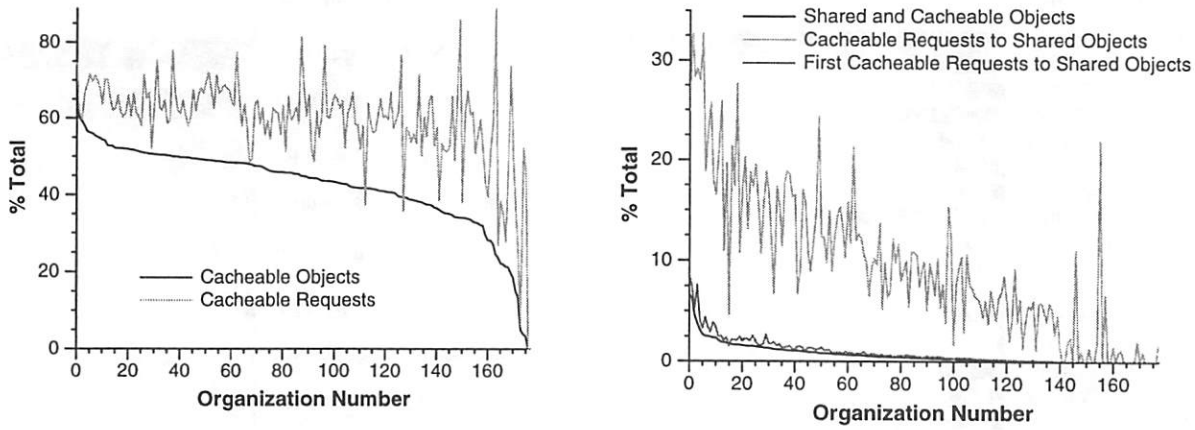


Figure 10: Breakdown by content-type of the uncacheable HTTP transactions.

- *Push-Content* – The content type "multipart/x-mixed-replace" is used by some servers to specify dynamic content.
- *Auth* – Requests that specify an Authorization header.
- *Vary* – Responses that specify a Vary header.

Figure 9 shows a breakdown of all HTTP requests, detailing the percentage that are uncacheable for each of the reasons listed above. As the figure shows in the bar labeled "Overall_Uncache", 40% of the requests are uncacheable for one or more of the itemized reasons. Queries and Response Status are the two major reasons for uncacheability. Adding up the percentages for each reason sums to an amount greater than the overall uncacheability rate, showing that many documents are uncacheable for more than one reason. The figure also shows, for each itemized reason, the percentage of HTTP requests that are uncacheable only due to that reason. Finally, the figure shows that 16% of Web requests are uncacheable for two or more reasons. Figure 10 shows the most common content types for the uncacheable documents.

Our intent in analyzing the cacheability of documents is to show which requests a deployed proxy cache would be allowed to store if it were given the request stream from our trace. However, one should not infer from our analysis that all of the uncacheable requests are truly dynamic content. Web content providers may choose to mark documents uncacheable for other reasons, such as the desire to track the behavior of individual users. Figure 10 shows that more than 12% of all the uncacheable



(a) Cacheable Objects and Requests

(b) Cacheable and Shared

Figure 11: The left graph shows the fraction of cacheable objects and cacheable requests accessed by each organization. The right graph shows the fraction of objects and requests that are both cacheable and shared by more than one organization.

documents have the image/gif content type, and we suspect that very few of these images are truly dynamic content.

Figure 11a shows, for each organization, the percentage of objects (black line) requested by the organization that are potentially cacheable. The light grey line shows, for each organization, the percentage of requests whose responses are cacheable. The figure shows that the percentage of cacheable objects is somewhat lower than the percentage of cacheable requests. The percentage of cacheable requests gives an upper bound on the hit rate each organization could see with an organization-local proxy cache.

Figure 11b shows, for each organization, the percentage of cacheable shared objects (the black line), and the percentage of cacheable shared requests in two categories. The medium grey line shows those first requests by an organization to globally shared objects. The light grey line shows the total number of requests by an organization to globally shared objects. The difference between these two lines represents the duplicate requests by an organization to globally shared objects. If each organization has its own cache, then the local cache can handle all duplicate requests whether or not there is a global cache. If there is a global cache in addition to the local caches, then the global cache will miss on the first request by any of the organizations, but will hit on all the first requests by other organizations that follow. One can conclude from this graph that there is significant sharing among organizations (as shown by the light grey line), but that a large fraction of that sharing is captured just with organizational caches (as shown by the difference between light and medium grey lines). Therefore, a global cache in addition to the local caches will help, but not nearly to the degree indicated by the amount of sharing among organizations. Another interesting question is whether a single global cache would be better than using local caches. We explore this question in a related paper [26].

A last factor that can affect the performance of caching is object expiration time. We found overall that only 9.2% of requests had an expiration specified. Most of these requests are to objects that expire quickly; 47% are to objects that expire in less than 2 hours. Interestingly, of those that did have an expiration specified, 26% had a missing or invalid date and 29% had an expiration time that had already passed.

Finally, we have not presented detailed cache simulations here; our objective is simply to analyze cacheability of documents in the most recent data. From our data, it appears that the trends with respect to cacheability of documents are getting worse. For example, our measurement that 40% of all document accesses are uncacheable is significantly higher than the 7% reported for client traces at Berkeley in 1997 [16]. Without widespread deployment of special mechanisms to deal with caching, such as caching systems that handle dynamic content [7, 8], the benefits of proxy caching are not likely to improve.

7 Conclusions

In this paper, we have collected and analyzed a large recent trace taken in a university setting. Our study has focused on sharing of Web documents within and among a diverse set of organizations within a large university.

We can reach the following conclusions from our data:

- Organization membership appears to be significant: members of an organization are more likely to request the same documents than a set of clients of the same size chosen at random from all the clients in the population. However, the vast majority of the requests made (and the objects requested) are to objects that are shared among multiple organizations.
- Objects that are simultaneously shared locally by an organization and globally with other organizations are more likely to be requested by an organization member than objects that are just shared locally or just shared globally. This suggests that the most-requested objects by an organization are globally and universally popular.
- The trace shows mostly minor differences relative to earlier traces in terms of many of the basic characteristics. However, we see two important differences compared to previous traces. The first is that the percentage of requests to uncacheable documents is significantly higher. The second is that a significant amount of audio/video content appears in our trace.

When analyzing these conclusions, one must keep in mind that we do not know how similar our university organizations are to typical commercial organizations that connect to the Internet, but we hope to investigate this question in future work. We have only begun to analyze the data we have collected. Other future work includes a more detailed statistical analysis of various aspects of the data already collected as well as a study of the evolution of WWW traffic characteristics over time. Towards this end, we plan to repeatedly trace and examine Web traffic at the University of Washington.

8 Acknowledgments

We would particularly like to thank Steve Corbato, Art Dong, Corey Satten, and the other members of the Computing and Communications organization at UW, who supported our effort. We also wish to thank Geoff Kuenning for his diligent shepherding that added greatly to the clarity of the paper. This research was supported in part by DARPA Grant F30602-97-2-0226, National Science Foundation grant EIA-9870740, US-Israel Binational Science Foundation grant 96-00247, and an IBM Graduate Research Fellowship.

References

[1] Jussara Almeida, Virgilio Almeida, and David Yates. Measuring the behavior of a World Wide Web server. Technical Report 96-025, Boston University, October 1996.

[2] Virgilio Almeida, Azer Bestavros, Mark Crovella, and Adriana deOliveira. Characterizing reference locality in the WWW. Technical Report 96-011, Boston University, June 1996.

[3] Martin F. Arlitt and Carey L. Williamson. Web server workload characterization: The search for invariants. In *Proc. of the ACM SIGMETRICS '96 Conference*, April 1996.

[4] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. Web caching and Zipf-like distributions: Evidence and implications. In *Proceedings of IEEE INFOCOM '99*, March 1999.

[5] Ramon Caceres, Fred Douglis, Anja Feldmann, Gideon Glass, and Michael Rabinovich. Web proxy caching: the devil is in the details. In *Workshop on Internet Server Performance*, June 1998.

[6] Pei Cao. Characterization of web proxy traffic and Wisconsin proxy benchmark 2.0, November 1998.

[7] Pei Cao, Jin Zhang, and Kevin Beach. Active cache: Caching dynamic contents on the web. In *Proc. of IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware '98)*, September 1998.

[8] Jim Challenger, Arun Iyengar, and Paul Dantzig. A scalable system for consistently caching dynamic web data. In *Proceedings of IEEE INFOCOM '99*, March 1999.

[9] Anawat Chankhunthod, Peter B. Danzig, Chuck Neerdaels, Michael F. Schwartz, and Kurt J. Worrell. A hierarchical Internet object cache. In *Proc. of the 1996 USENIX Technical Conference*, January 1996.

[10] Mark E. Crovella and Azer Bestavros. Self-similarity in World Wide Web traffic: Evidence and possible causes. In *Proc. of the ACM SIGMETRICS '96 Conference*, April 1996.

[11] Carlos R. Cunha, Azer Bestavros, and Mark E. Crovella. Characteristics of WWW client-based traces. Technical Report BU-CS-95-010, Boston University, July 1995.

[12] Fred Douglis, Anja Feldmann, Balachander Krishnamurthy, and Jeffrey Mogul. Rate of change and other metrics: a live study of the World Wide Web. In *Proc. of the USENIX Symposium on Internet Technologies and Systems*, November 1997.

[13] Brian Duska, David Marwood, and Michael J. Feeley. The measured access characteristics of World Wide Web client proxy caches. In *Proc. of the USENIX Symposium on Internet Technologies and Systems*, November 1997.

[14] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder. Summary cache: A scalable wide-area web cache sharing protocol. In *Proceedings of ACM SIGCOMM '98*, August 1998.

[15] Anja Feldmann, Ramon Caceres, Fred Douglis, Gideon Glass, and Michael Rabinovich. Performance of web proxy caching in heterogeneous bandwidth environments. In *Proceedings of IEEE INFOCOM '99*, March 1999.

[16] Steven D. Gribble and Eric A. Brewer. System design issues for Internet middleware services: Deductions from a large client trace. In *Proc. of the USENIX Symposium on Internet Technologies and Systems*, November 1997.

[17] James Gwertzman and Margo Seltzer. The case for geographical push caching. In *Proc. of the Fifth Annual Workshop on Hot Operating Systems*, May 1995.

[18] P. Krishnan and B. Sugla. Utility of co-operating web proxy caches. In *Proc. Seventh InternationalWorld Wide Web Conference*, April 1998.

[19] Thomas M. Kroeger, Darrell D. E. Long, and Jeffrey C. Mogul. Exploring the bounds of web latency reduction from caching and prefetching. In *Proc. of the USENIX Symposium on Internet Technologies and Systems*, November 1997.

[20] Michal Kurcewicz, Wojtek Sylwestrzak, and Adam Wierzbicki. A distributed WWW cache. In *3rd International WWW Caching Workshop*, June 1998.

[21] Bruce A. Mah. An empirical model of HTTP network traffic. In *Proceedings of IEEE INFOCOM '97*, April 1997.

[22] Steve McCanne and Van Jacobson. The BSD Packet Filter: A new architecture for user-level packet capture. In *Proc. of the USENIX Technical Conference*, Winter 1993.

[23] Jeffrey C. Mogul. Network behavior of a busy web server and its clients. Technical Report 95/5, Digital Equipment Corporation Western Research Laboratory, October 1995.

[24] Michael Rabinovich, Jeff Chase, and Syam Gadde. Not all hits are created equal: Cooperative proxy caching over a wide area network. In *3rd International WWW Caching Workshop*, June 1998.

[25] Squid internet object cache, http://squid.nlanr.net.

[26] Alec Wolman, Geoffrey M. Voelker, Nitin Sharma, Neal Cardwell, Anna Karlin, and Henry M. Levy. On the scale and performance of cooperative web proxy caching. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (To Appear)*, December 1999.

[27] Lixia Zhang, Sally Floyd, and Van Jacobson. Adaptive web caching. In *Proc. of the 1997 NLANR Web Cache Workshop*, June 1997.

The Ninja Jukebox

Ian Goldberg, Steven D. Gribble, David Wagner, and Eric A. Brewer
The University of California at Berkeley
{iang,gribble,daw,brewer}@cs.berkeley.edu

Abstract

We present the design and implementation of the "Ninja Jukebox", an infrastructural service that allows a community of users to build a distributed, collaborative music repository that delivers digital music to Internet clients, and that performs simple collaborative filtering based on users' song preferences inferred by the service. The Jukebox, implemented in Java, was designed to allow rapid service evolution and reconfiguration, simplicity in participation, and extensibility. We demonstrate that our careful use of a distributed component architecture enabled rapid prototyping of the service, and that our use of carefully designed, strongly typed interfaces enabled the smooth evolution of the service from a simple prototype to a more complex, mature system.

1 Motivation

The Internet is evolving towards a *service infrastructure:* a network of rich, robust, and often professionally maintained services that are conveniently accessible to people through the web. However, the fact that these services rely on the web to present their content effectively restricts their users to be human; the lack of structure and well-defined types in web content makes it all but impossible for computer programs to interact with most Internet services, despite the obvious benefits of being able to do so (such as service composition, richer search and information access services, etc.). Several recent efforts have attempted to introduce such structure and typing to the web, such as the WIDL [12] and WebL projects [16], or the ongoing W3C XML developments [6].

The UC Berkeley Ninja project[1] is pursuing a complementary path to these efforts: we are building an infrastructure for supporting fault tolerant, highly available, scalable services composed of a number of well-circumscribed components, each of which exports a strongly-typed, programming-language level interface [10] accessible using RPC-like mechanisms [4]. Explicitly exposing service interfaces and making use of strong typing has a number of benefits, including forcing authors to carefully design the boundary between their services and the rest of the world, making those services accessible to programs, and allowing the composition of services by infrastructural elements. We believe that when a large number of such services are deployed, a network-externality effect will occur, causing the power of an individual service to be greatly enhanced by interaction with the many other available services.

In this paper, we describe one such service: the *Ninja Jukebox.* The Jukebox allows a community of users to collaboratively build a distributed music repository out of both music CDs and MP3 files stored in local filesystems, and to use simple collaborative filtering to allow individual users to filter their music preferences according to other community members' explicit and implicit recommendations. In section 2, we discuss the design rationale that went into the Ninja Jukebox, and reflect on how the Ninja project's service philosophy influenced this design. Section 3 describes our Java-based Jukebox implementation and how we smoothly evolved it, and section 4 presents some of the limitations of our implementation and the lessons we learned while building it. Finally, in section 5, we present related work.

2 Design Philosophy

The Ninja Jukebox application was originally conceived of to "scratch the itch" of several graduate students: to be able to harness the large number of unused CD-ROM drives in the 100+ node Berkeley network of workstations (NOW) [3] and present a single, unified view of all music in all drives. Over time, the Jukebox has vastly evolved in complexity and richness. It now transparently supports both raw audio CDs in CD-ROM drives and MP3 files in local filesystems, and it performs authentica-

[1] Project home page: `http://ninja.cs.berkeley.edu`

tion and access control in order to adhere to copyright laws. It exports both a programmatic interface and an HTML interface for backwards compatibility with browsers; its programmatic interface includes a collaborative filtering service that deduces users' song preferences, and allows one to construct song playlists based on simple boolean combinations of other users' preferences.

The Ninja Jukebox was designed with several specific goals in mind. The first goal was that the Jukebox should be a communal, collaborative service. Individuals should be able to add or retract their personal collection of music from the Jukebox as they please, without requiring special intervention from a centralized administrator. This implies that contributors should be given as flexible as possible of a "service contract"—they must be allowed to retain control over their own contributions, while still ensuring that the overall Ninja Jukebox service maintains as stable as possible of a view to the rest of its users. The Jukebox service therefore must be able to adapt to changing group membership by gracefully masking unpredicted failures or disappearances.

Another goal was for the Jukebox service to retain flexibility, extensibility, and the facility for rapid evolution. As the evolution of the Jukebox has explicitly demonstrated, applications are not cast in stone, and services should not remain immutable once they have been released and are in use by applications and users. We therefore wanted our infrastructure to admit the evolution of its services, and we wanted to design the Jukebox service in such a way as to most easily allow it to be extended in unforeseen ways.

2.1 Design Implications

In order to meet the above goals, we made the following three explicit design decisions: the adoption of a distributed component architecture to decompose the Jukebox service into a small number of carefully chosen, functionally decoupled pieces, the imposition of a rich, strongly typed interface on these components (including carefully chosen data structures that precisely describe the contents of the Jukebox), and the use of soft state to achieve eventual consistency in the Jukebox.

Disciplined use of a distributed component architecture: as exemplified by Sun's Jini [21] and Corba [20], distributed component architectures advocate the use of an object-oriented language to decompose applications into smaller, self-contained objects, and the distribution of those objects across machine boundaries, relying on mechanisms such as RPC to perform inter-object communication. Component architectures make it simpler to begin with and maintain a clean design throughout the service's lifetime: the separation into objects allows for a separation of concerns, a tenet of good software engineering.

In the Ninja Jukebox, we decomposed our service into three major components, each respectively responsible for: (1) managing local collections of music (independent of physical and logical format), (2) the integration of many such collections of music and the maintenance of metadata about the music (such as users' song preferences), and (3) the client-side retrieval and playing of music from the service. This deliberate decomposition is what ultimately allowed the Jukebox to evolve so painlessly—each component's functionality is well encapsulated and isolated from other components, meaning that these components can internally evolve without affecting the rest of the system, and that new components can be added that compose with existing pieces to enhance the overall service. For example, the component responsible for managing local collections of music encapsulates information and access mechanisms particular to a music format, and thus the transition supporting only audio CDs in CD-ROM drives and also supporting MP3 files stored in a file system merely involved introducing a subclass of that component. Similarly, we could envision adding subsequent subclasses that would contain all music available from popular music web sites (such as `http://www.mp3.com`), or would serve as a gateway to receive music broadcasts (such as MBONE vat sessions).

Strongly-typed interfaces: in our opinion, the use of a distributed component architecture is only a partial step towards a properly decomposed service: the careful design of the interfaces between those components is a second, crucial step. An interface to a component is a declaration of both syntax and semantics, and as such is a contract that binds the component author to maintain those semantics even when the component is enhanced or extended through subclassing. Furthermore, the API to the service ultimately dictates the expressive power that clients of that service have available to them. We believe that an infrastructure service is defined by its interface and a declared set of guarantees about its performance and availability.

In the Jukebox, our APIs include data structures that richly describe content. These structures enable intelligent applications such as clients that group music on arbitrary terms, or that allow users

to construct playlists based on either explicit declarations or inferred preferences gleaned from the service's observation of their listening history. This focus on strongly-typed interfaces helps remove barriers to rapid service evolution by forcing service authors to carefully design and explicitly declare each of their components' interfaces, and therefore their implied service contracts.

Use of soft state to achieve eventual consistency: as a side-effect of making the Jukebox collaborative, we could not rely on any particular person's contributions to remain available. We thus designed the infrastructure so that a contributor periodically announces the presence of his/her music to a common master repository in order to add music to the overall Jukebox. The act of a person adding music to the Jukebox is therefore treated as a hint rather than a promise: components cannot rely on that music being there, and they must gracefully handle the case in which a particular song abruptly becomes unavailable. We also treat entries in the master repository as a lease, and expire them if the periodic announcements stop. The master repository correspondingly contains an approximate view of all available music; this view continually approaches the correct view over time. This leased approach is also used in our authentication mechanisms: when a client requests a song from the Jukebox, it must first authenticate itself, the result of which is a capability that is good for a single use or for thirty seconds: the Jukebox components lazily time out these capabilities as necessary.

Not all state in the Jukebox is soft-state: users' song preferences, for example, are stored as hard state by dedicated, highly-available infrastrucure in what we call a "base"[10]. A base is composed of everything needed to build an available compute cluster, including system administration, a secure machine room, redundant networks, UPS, etc., and as such is an ideal environment in which to protect hard state.

3 Implementation

This section of the paper describes the implementation of the Ninja Jukebox service and client, and their evolution through three stages of functionality. The first version of the service only supported the playback of raw audio CDs from the CD-ROM drives of Jukebox servers. In the second version of the service, we added the ability to convert raw CDs into compressed MP3 files, and for those MP3 files to be played over the network; this second version also included authentication and access control mechanisms to enforce copyright protection. Finally, we added a simple collaborative filtering mechanism to the third and current version of the Jukebox.

We chose to implement the Ninja Jukebox service in Java, both because Java trivially enables distributed components and because the Ninja project has developed a significant amount of infrastructure in Java. This infrastructure includes authenticated remote method invocation (RMI) and a cluster-based service platform called "MultiSpace" [10] that was designed to support scalable and rapidly evolvable infrastructure services.

3.1 Ninja Jukebox v1.0: raw audio CD playback

As shown in Figure 1, the first version of the Ninja Jukebox implementation was decomposed into the following elements:

The SoundSmith: SoundSmiths are responsible for indexing and maintaining a structured collection of music. The version 1.0 SoundSmith indexes music on an audio CD in a local CD-ROM drive, making use of a service that acts as an HTTP to RMI gateway to provide programmatic access to an online "CDDB" database[15]. This database provides a mapping from a CD's track timing information to detailed information about the CD's author, song titles, and song durations. SoundSmiths periodically send beacons to the MusicDirectory; through these beacons, they both announce their existence to the MusicDirectory and present the list of songs that they maintain. Anyone that wishes to contribute music to the Jukebox must only run a SoundSmith that can index and serve that music. SoundSmiths can be started-up and torn down at any time, as each SoundSmith is completely autonomous, and the beacons emitted by the SoundSmith are treated as soft-state by the MusicDirectory. A SoundSmith serves music by streaming it off of an audio CD from the CD-ROM drive of an infrastructure workstation and transmitting it in uncompressed .au format through an (untyped) HTTP interface.

The MusicDirectory: As previously mentioned, SoundSmiths periodically beacon their existence and a list of their music to the MusicDirectory. The role of the MusicDirectory is to keep track of these beacons, and to build up an integrated list of all available music and of all running SoundSmiths. Clients use the MusicDirectory as a level of indirection that shields them from needing to inde-

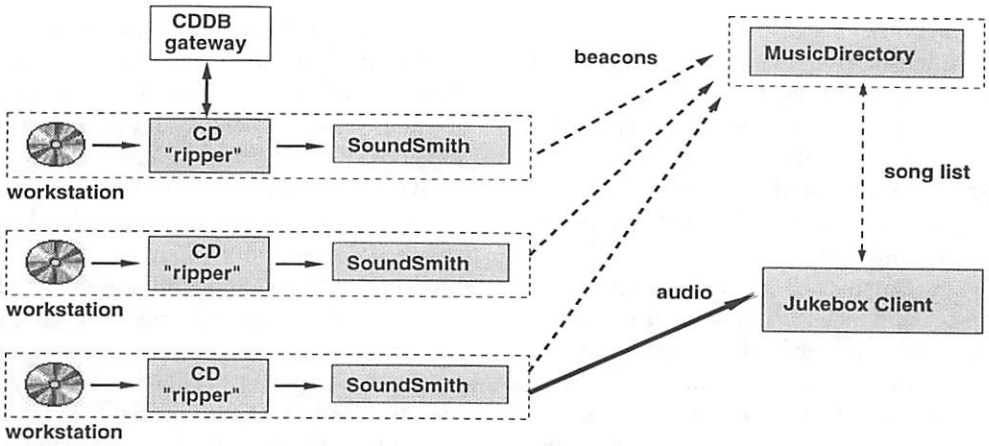


Figure 1: The Ninja Jukebox v1.0 architecture

pendently discover the location of all SoundSmiths in the Jukebox. Ultimately, this centralized MusicDirectory limits the scale of a Jukebox, since all SoundSmiths repeatedly send it listings of music.

Jukebox Clients: Jukebox Clients interact with a MusicDirectory to gather a listing of available music, and with many SoundSmiths to receive and play specific songs. We have currently implemented two clients. The first presents a graphical user interface to the user (figure 2), and allows users to build playlists of available songs. Music streamed to this client is shuttled to external music players that understand many music formats and have the ability to play music as it is streamed over the network. Internally, this client is decomposed into a GUI front end and a song selection back end. The GUI front end provides the user with controls for constructing playlists, and with familiar *play*, *stop*, *pause*, *fast-forward*, and *reverse* buttons. The song selection back end selects specific songs to play given the list of currently available music from the MusicDirectory, the user's manually constructed playlist, and events that are generated when the buttons such as *play* or *stop* are pressed. The second client is a proxy that converts between the APIs and data structures exported by the Ninja Jukebox service and HTML forms. This proxy allows conventional HTML browsers to access the Jukebox; music is streamed through the proxy to the browser, or presumably to the browser's helper applications that can actually understand specific audio formats.

This first version of the Jukebox service was well received even though it suffered from a number of drawbacks. The fact that all audio was transmitted in an uncompressed format resulted in excessive traffic on our local networks, greatly limiting

Figure 2: The Ninja Jukebox client GUI

the number of clients that could simultaneously access the Jukebox. Furthermore, the fact that music could only be served from audio CDs physically present in CD-ROM drives limited the amount of music that could be present in the Jukebox at any given time, since we had a limited (although large) number of CD-ROM drives at our disposal. Finally, the lack of any security infrastructure prevented us from widely releasing the Jukebox service and client, even within our department, since it would become trivial for users to violate copyright protection legistlation, either accidentally or deliberately.

3.2 Ninja Jukebox v2.0: MP3 playback and security

The separation of the Jukebox into the previously described components satisfied our primary design goal: to construct a collaborative service, in which anyone can contribute their collection of music to the Jukebox. A second design goal was to allow for the evolution of the service; in order to test this goal (and to satiate the demands of the clients of the v1.0 Jukebox), we extended the Jukebox functionality to

produce the v2.0 version of the service. This version of the service attempted to overcome the drawbacks of the v1.0 prototype by including two new major features: the transparent support of MP3 files, and support for access control and client authentication.

We also slightly modified the Jukebox by having SoundSmiths only report their existence to the MusicDirectory rather than the complete list of music that they manage; in the v2.0 infrastructure, clients discover SoundSmiths through the MusicDirectory, but then ask each individual SoundSmith for its list of locally available music. This modification drastically reduced the size of the SoundSmith's beacons, which eased the scaling bottleneck caused by the centralized MusicDirectory. This bottleneck became increasingly evident as the body of music stored in the Jukebox grew to over 4,400 songs (375 albums, accounting for more than 25 gigabytes of hard drive space and 320 hours of music).

3.2.1 MP3 Support

MP3 support was surprisingly easy to add to the Jukebox service. To do it, we simply created a subclass of the SoundSmith component that understood how to index and stream MP3 files on a regular filesystem instead of audio tracks from an audio CD in a CD-ROM drive. The data structures embedded in the SoundSmith's beacons are only metadata, and as such are totally independent of the specific format in which the music is actually kept. In order to play music, Jukebox clients interact with the MusicDirectory service to fetch an HTTP URL for a song; this URL is served by the SoundSmith that maintains the song. Because the song data is streamed to external music player software that happens to understand MP3 formatted music, the Jukebox clients never need to understand anything about the music format. When we deployed several of these MP3-aware SoundSmiths in our infrastructure, Jukebox clients suddenly became aware of a much larger set of available music, and transparently began accessing the newly available MP3 files.

The MP3 files maintained by SoundSmiths are created by helper daemons that batch convert music CDs to MP3 formatted files by first "ripping" raw audio from the CD, and then compressing that raw audio into an MP3 file and its associated artist and album metadata (figure 3). These daemons run in the background on all of our Jukebox workstations, effectively crawling the Jukebox for new music to MP3 compress and add to the Jukebox. While this conversion is happening, an audio CD SoundSmith can serve the music directly off of the audio CD;

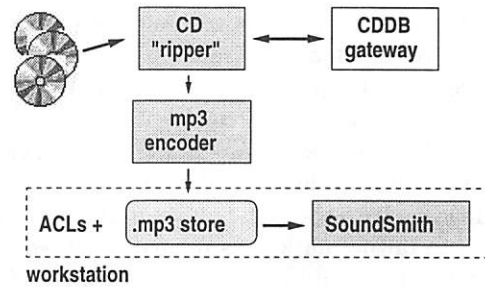


Figure 3: MP3 Support in the Jukebox v2.0

after the conversion is finished, the music can be served in the preferable MP3 format.

We attribute the ease with which we added support for MP3 files to the Jukebox infrastructure to our use of a distributed object infrastructure and to the strongly-typed interfaces between our Jukebox components. The ability to subclass in order to specialize the SoundSmith allowed us to maintain its RMI interface, and thus upgrade its functionality in a manner that was transparent to the rest of the Jukebox. Transparency was meaningful because of the presence of explicit interfaces between the components; achieving transparency in this case was a manner of maintaining both the syntax and declared semantics of the interface.

3.2.2 Security Infrastructure

For the Jukebox, the only relevant security issues are access control and authentication. Our authentication mechanism is based on SecureRMI, a variant of RMI—Java's standard remote method invocation protocol [17]—that we have developed to operate over a cryptographically-secured channel. With this tool in place, the access control problem becomes relatively easy: for each song in a SoundSmith, that SoundSmith maintains an ACL (a list of SecureRMI principals allowed to play that song). The access control mechanism thus is as simple as having the SoundSmith look up an entry in a list. The SoundSmith also hands out capabilities to authenticated principals that allow them to access specific songs for a limited amount of time: these capabilities are good for a single access, and expire if not used within 30 seconds. Note that the MusicDirectory does not need to authenticate the identity of clients, as it is entrusted only with a list of available songs and SoundSmiths, and not the songs' content. For the proxied HTML-based client to work, however, the proxy itself must be entrusted with its users' credentials, since HTML browsers do not

have the ability to interact with our SecureRMI infrastructure directly.

Currently, our policy for access control is relatively simplistic: a principal can only listen to a copyright-protected song if she has previously demonstrated knowledge of the song contents (e.g. by uploading it to the Jukebox); unrestricted access is given to music marked as non-copyrighted. This approach is inspired by legal considerations: if people can't abuse the Jukebox to gain access to music they don't already have, it seems unlikely that the Jukebox will be accused of violating copyright laws. However, the Jukebox could also accommodate more sophisticated policies for access control, such as support for group ownership where only one group member can listen to a song at a time, or a pay-per-use scenario under which royalties could be collected and submitted to the copyright holders. The flexibility of our design makes such variations on authentication quite straightforward.

Returning to the authentication mechanism, SecureRMI was the piece of the security architecture that demanded the vast majority of our security engineering effort. SecureRMI (optionally) authenticates the endpoints and then encrypts the remainder of the communication with a Triple-DES session key derived from a Diffie-Hellman key exchange. We also provide a certification infrastructure for endpoint public keys and tools for managing them; certificates bind the service's fully-qualified class name (or the client's identity) to the server's (or client's) public key.

Of course, there is nothing new about the concept of establishing a secure channel with the use of encryption [18, 22]. However, we feel that our implementation may be of interest primarily because it exists: we are not aware of any other free, Java implementation with similar functionality.[2]

One novel feature of our SecureRMI is that it provides transparent support for a very broad range of "authentication" technologies. We have abstracted away many of the irrelevant details of the algorithms to build a very general model of authentication. For instance, public-key authentication is implemented in `DSAAuthenticator` and `DSAVerifier`, which are subclasses of the generic `Authenticator` and `Verifier` classes; SecureRMI only references the generic `Authenticator/Verifier` superclasses, so it is ignorant of the details of their implementation. This architecture is very flexible: after the core infrastructure was in place, we later added a symmetric-key challenge-response protocol with about two days of coding.

As a result, extending the Jukebox to support pay-per-use access will require only minimal effort. We would just add a `PayPerUseAuthenticator` that, instead of sending a public-key signature for authentication, sends a digital coin. This is a direct result of our design goal that services be easy to evolve and extend.

Our general model of authentication also allows each collaborator to specify her own access-control policies for the music she serves; one SoundSmith could be serving music on an ACL basis, another could be serving only free music, but only to hosts in a certain domain, and others could be charging various amounts to listen to the audio stream. The flexibility provided by this mechanism further enhances the communal, collaborative nature of the Jukebox, by removing access-control policy from any central authority.

3.3 Ninja Jukebox v3.0: the collaborative DJ service

Most recently, we have extended the Jukebox service to provide song selection based on inferred user preferences as well as some simple collaborative filtering functionality. In the v1.0 and v2.0 Jukebox services, song playlists are manually constructed by users and successive songs to be played are chosen from these playlists by simple random selection. Our collaborative filtering extension refines this selection with an infrastructural "DJ" service that exploits individual and collaborative song preferences.

A key observation is that an infrastructure service may, over time, learn user song preferences by observing UI events. Songs that the user always "fast-forwards" past are probably songs the user doesn't like; in contrast, listening to a song until its completion may be an indication that the user enjoyed the song. This observation forms the basis for our preference inference mechanism. As we described in section 3.1, our graphical Jukebox client is decomposed into a GUI front end and a song selection back end. In our v3.0 Jukebox infrastructure, we have decoupled this song selection from the client executable, moving it instead into the network as a infrastructure service so that our song selection algorithms may be upgraded and evolved transparently without modifying code on the client side. This enabled us to extend the original unintelligent song selection algorithm by interposing on the selection interface.

In our DJ implementation, a rating storage ser-

[2] JDK 1.2 includes hooks so you can encrypt RMI communications with SSL, if you have a SSL library; but we do not know of any free SSL implementations for Java [8].

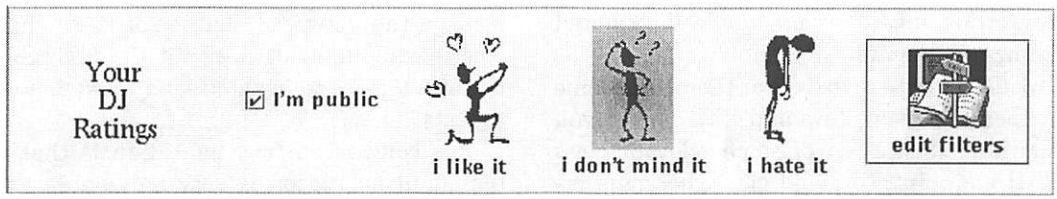


Figure 4: The DJ collaborative filtering client GUI element

vice in the infrastructure subscribes to client UI events; every time a user presses a button such as "fast-forward", a SecureRMI call is made into this DJ service to report the event. The DJ interprets these events as implicit hints about the user's song preferences, and updates a persistent database[3] on disk to reflect the new information about the user. Our prototype also allows users to explicitly specify their preferences about individual songs, if they like. Still, the advantage of transparent preference inference is that it requires no extra action on the part of the user.

A second key observation is that, when preferences for many users are all stored together in the infrastructure, there is a great opportunity to mine this data for cross-user information and to provide collaborative services [19]. We have implemented a simple collaborative filtering application for the DJ. By default, a user's preferences are regarded as private and are stored securely in the infrastructure, with no access allowed to third parties; however, we allow users to publicly export read-only access to their preferences to other users. Marking one's preferences as public allows one to share preferences between multiple users. For example, our implementation allows a user to temporarily use someone else's preferences for song selection (assuming, of course, that those preferences have been explicitly marked as public). More interestingly, a user may combine the preferences of multiple other public users and use the result to drive the Jukebox client's song selection algorithm. This is a useful way to accomodate multiple listeners with different preferences; for example, in an shared environment in which several students occupy the same office, a useful combination would be to play songs that are in the intersection of the students' sets of likable songs.

The DJ extensions to the core Jukebox service resulted in minimal changes to the existing codebase; rather, the extensions were mostly encapsulated within the new DJ component that was added to the Jukebox infrastructure. The required changes to the existing codebase were limited to modifications to the Jukebox client's song selection algorithm to request a playlist from the DJ service, and to the enhancement of the Jukebox client front end to send a copy of all relevant events to the appropriate rating storage service. We also augmented the Jukebox client GUI to include controls that allow the user to explicitly indicate preferences for specific songs (figure 4).

[3]We actually used a distributed, persistent hash table to keep track of user preferences. This hash table (described in [9]) is partitioned and replicated across nodes in a dedicated workstation cluster, and provides the DJ fault-tolerant access to the persistent user preferences data.

4 Discussion

In this section of the paper, we first present several lessons that we learned about using Java as a service construction language, and then we discuss several limitations of the current Jukebox implementation.

4.1 Java as a Service Construction Language

We were surprised to find that the decision to use Java as a rapid prototyping tool met with mixed results. Certainly, Java's high-level programming model made for extremely rapid prototyping: the first version of the Jukebox service was built in 2 days by a team of 3 students. Java's strong typing also encouraged modularity, which made it easier to extend and evolve the service several times: once to migrate from playing CDs in real-time towards serving as a shared MP3 repository, later to extend the service to add a security model, and a third time to transparently learn song preferences and to add support for collaborative filtering. In all three cases, strong typing helped assure the separation between client code (which should change rarely) and networked services (which may evolve frequently) that was a key ingredient to success. Also, the tight coupling of RMI with Java, and the existence of the

Ninja SecureRMI infrastructure made distributed programming less painful.

What we didn't anticipate is that there were some negative aspects to using Java and RMI. When you change the implementation of some relevant class on one RMI endpoint, to avoid class checksum errors you must grab the new source code and recompile on all other endpoints too. Thus, updates to the service code require synchronized updates at all RMI endpoints, which is an administrative annoyance. Moreover, though we didn't realize it at first, if we had used RMI for all of our external service interfaces, the situation would have been far worse: each upgrade to the Jukebox service would potentially have required the clients to be updated too, a terrible scenario for service evolution! Fortunately, we got lucky: most of our external interfaces that changed used HTTP, not RMI. Our interpretation (in retrospect) is that we *should* have done a better job of picking strongly-typed interfaces to the outside world (rather than having any untyped HTTP connections) and frozen these interfaces from the outset, but we didn't. We gained considerable leverage from the use of narrow, strongly-typed interfaces between internal Jukebox components, and if we were to re-implement the Ninja Jukebox, we would strive to do the same for all of our external interfaces as well.

4.2 Limitations

Our current prototype of the Jukebox service has a number of limitations. First, the Jukebox is not intended—in its current incarnation—as a wide-area distributed service. Instead, we have focused on providing service within a single organization. As an example, the MusicDirectory service is currently centralized, which means that it would quickly become a bottleneck if we moved to a wide-area usage scenario. We have also made the simplifying assumption that all nodes in the system are relatively close to each other (in terms of network latency and bandwidth), so that from the client's point of view all SoundSmiths are created equal.

Although a wide-area Jukebox service would be limited by the capacity of the underlying network, with some more work we could extend Jukebox to address wide-area concerns. Two changes would be required: (1) the MusicDirectory service would have to become wide-area aware, using standard techniques such as replication, caching, and aggregation to distribute song listings around the world, and (2) we might want to replicate MP3's across the wide-area, using pre-fetching and caching to reduce the load on the network. Neither of these changes are conceptually difficult; we built the Jukebox because it was a service we wanted, and so we ignored these aspects.

A second important limitation is that our current implementation is very naive about multimedia operations. The MP3 data is transmitted over a HTTP connection, and thus inherits all of the problems of TCP for multimedia data: no quality-of-service guarantees, potentially high latency, unwanted buffering, and so on. There is also no rate limiting; we merely blast as fast as we can, which runs the risk of overloading the network. Nor are our clients particularly sophisticated about multimedia issues: our MP3 player doesn't do real-time scheduling, so during heavy paging we occasionally experience playback glitches. Nonetheless, these issues are largely orthogonal to our research; instead, we focused on testing the hypothesis that we can rapidly build a highly evolvable service if we carefully use component architectures and strongly typed interfaces, and thanks to this extensibility we believe a future version of the Ninja Jukebox could easily include better multimedia delivery technology.

Thirdly, our current prototype has poor performance for the Java security operations. Right now, we are using a pure-Java cryptographic library, with no JIT, and as a result the public-key operations are very CPU-intensive: the initial SecureRMI handshake currently takes about 4 seconds to complete. Of course, these numbers could be dramatically reduced by any of many techniques (native code, precomputation, caching, session-reuse, etc.), but so far the performance impact has been relatively innocuous.

5 Related Work

Keeping collections of audio files in a net-accessible way is obviously not a new idea. The simplest way to publish one's music collection to the net is just to make it accessible as a WWW or FTP archive. Many people do this today, but the utility of unconnected collections of audio is low. In order to make these collections more useful, dedicated MP3 search engines such as `mp3.lycos.com` have appeared. These search engines try to be your "one-stop shopping" for MP3's, by telling you where on the Internet you can find your favorite pirated songs.[4] More recently, commercial jukebox prod-

[4]Lycos itself, of course, does not illegally publish copyrighted material.

ucts have become available that allow you to organize and play locally-stored MP3's, but these products typically do not permit sharing between users, nor do they offer collaborative or interactive features.

This simplest kind of jukebox system is missing a number of benefits that the Ninja Jukebox provides. Simple directories of MP3 files offer no cohesive framework for security-related features such as authenticated or pay-per-use access. In addition, our component architecture allows the SoundSmiths to be active and easily updatable participants in the transmission of the audio, as opposed to merely serving a static file. This allows features such as transparent format conversion (`.wav` files on file systems, raw audio on CD-ROM drives, or MP3 files on file systems) and support for multiple transport mechanisms (streamed audio over HTTP, or VAT audio over a multicast IP channel).

Another approach has recently come from SHOUTcast [11]. SHOUTcast is an "Internet radio" system that allows a site to serve an audio stream that can be picked up by multiple clients. The clients have to listen to what the SHOUTcast servers decide to play; they have no way to interact with the servers. Although each SHOUTcast server offers the same real-time audio stream to each of its clients, and though its name would imply something more clever, its underlying technology is just multiple simultaneous unicasts of the same data. SHOUTcast servers communicate with one or more central databases in order to register the names of the programs they are currently "broadcasting". These databases can be queried by client programs (like MP3Spy [13]) to allow users to choose what channels they would like to hear.

The largest difference between SHOUTcast and our work is that our goal was to provide a communal, collaborative, interactive jukebox, as opposed to a passive Internet radio station. That having been said, however, it would be possible for a SoundSmith to transmit any particular song over a true multicast channel [7]. SHOUTcast servers also do no user authentication; one might indeed imagine that an Internet radio service would have no need for such a thing. However, given the broad view of "authentication" taken by the Ninja Jukebox, one could see that implementing, for example, subscription-based access or pay-per-use access, could add value (better quality of service, for example) even to a non-interactive service like Internet radio.

A related approach is the Interactive Multimedia Jukebox [1, 2], a system that allows one to add a measure of interactive preference feedback to traditional broadcast paradigms.

More recently, the SDMI project is starting to tackle the issues associated with copyright control and rights management, using a combination of tamperproof hardware (or software!) on the client end as well as watermarking and other technologies [14]. We view SDMI as largely orthogonal to our work: we have focused on building a music delivery service, rather than on what is done after the music has been delivered.

There have been a number of projects involved in the delivery of audio and/or video over digital networks (for example, [5]); these projects mainly concern themselves with the technology of media delivery. In contrast, we have left that issue largely unaddressed, as it is orthogonal to our own goals; we were more interested in the mechanisms of the service, rather than the mechanisms of serving.

6 Conclusions

In this paper, we demonstrated that Java is a convenient language for the construction of infrastructural services, although there are several pitfalls and hurdles (such as performance, vagaries about the internals of its RMI facilities, etc.) that need to be addressed or avoided in order to successfully build such services. We also partially validated our hypothesis that infrastructural services which explicitly expose a strongly typed, programmatic API (as opposed to an unstructured interface designed only for humans) are conducive to the construction of complicated applications. Finally, we demonstrated that a distributed component architecture enabled the rapid development of an infrastructural Jukebox service, and that through the careful decomposition of the service into components and deliberate attention given to the design of the service's internal and external interfaces, we were able to smoothly evolve the first generation Jukebox into a more rich and mature service.

References

[1] K. Almeroth and M. Ammar. The Interactive Multimedia Jukebox (IMJ): A New Paradigm for the On-Demand Delivery of Audio/Video. In *Seventh International World Wide Web Conference (WWW-7)*. World Wide Web Consortium, April 1998.

[2] K. Almeroth and M. Ammar. An Alternative Paradigm for Scalable On-Demand Applications:

Evaluating and Deploying the Interactive Multimedia Jukebox. *IEEE Transactions on Knowledge and Data Engineering Special Issue on Web Technologies*, July/August 1999.

[3] Thomas E. Anderson, David E. Culler, and David Patterson. A Case for NOW (Networks of Workstations). *IEEE Micro*, 12(1):54–64, February 1995.

[4] Andrew D. Birrell and Bruce Jay Nelson. Implementing Remote Procedure Call. *ACM Transactions on Computing Systems*, 2(1):39–59, February 1984.

[5] William J. Bolosky, Joseph S. Barrera III, Richard P. Draves, Robert P. Fitzgerald, Garth A. Gibson, Michael B. Jones, Steven P. Levi, Nathan P. Myhrvold, and Richard F. Rashid. The Tiger Video Fileserver. Technical Report MSR-TR-96-09, Microsoft Research, Advanced Technology Division, April 1996.

[6] The World Wide Web Consortium. Extensible Markup Language (XML) version 1.0. `http://www.w3.org/XML`, Feb 1998.

[7] Stephen E. Deering, Deborah Estrin, Dino Farinacci, Van Jacobson, Ching-Gung Liu, and Liming Wei. An Architecture for Wide-Area Multicast Routing. In *Proceedings of SIGCOMM '94*, University College London, London, U.K., September 1994.

[8] Li Gong. New Security Architectural Directions for Java. In *Proceedings of IEEE COMPCON*. IEEE, February 1997.

[9] Steven D. Gribble. Simplifying Cluster-Based Internet Service Construction with Scalable Distributed Data Structures. Ph.D. Qualifying exam, available at `http://www.cs.berkeley.edu/~gribble/papers/quals/sdds-cluster.ps.gz`, April 1999.

[10] Steven D. Gribble, Matt Welsh, Eric A. Brewer, and David Culler. The MultiSpace: an Evolutionary Platform for Infrastructural Services. In *Proceedings of the 1999 Usenix Annual Technical Conference*, Monterey, California, USA, Jun 1999.

[11] Nullsoft Inc. SHOUTcast Service home page. `http://www.shoutcast.com/`.

[12] WebMethods Inc. WIDL—Web Interface Description Language. *World Wide Web Journal*, 1997. Special issue—XML: Principles, Tools, and Techniques.

[13] Game Spy Industries. The MP3Spy Client home page. `http://www.mp3spy.com/`.

[14] Secure Digital Music Initiative. SDMI Portable Device Specification, part 1, version 1.0, July 1999. `http://www.sdmi.org/`.

[15] Ti Kan and Steve Scherf. CDDB Specificaton. `http://www.cddb.com/ftp/cddb-docs/cddb_howto.gz`.

[16] Thomas Kistlera and Hannes Marais. WebL—A Programming Language for the Web. In *Computer Networks and ISDN Systems (Proceedings of the WWW7 Conference)*, volume 30, pages 259–270, Brisbane, Australia, April 1998.

[17] Sun Microsystems. Java Remote Method Invocation—Distributed Computing for Java. `http://java.sun.com/`.

[18] R.M. Needham and M.D. Schroeder. Using Encryption for Authentication in Large Networks of Computers. *CACM*, 21(12):993–999, December 1978.

[19] Firefly Network. Firefly Passport. `http://www.firefly.net`, 1995.

[20] The Object Management Group (OMG). The Common Object Request Broker: Architecture and Specification, February 1998. `http://www.omg.org/library/c2indx.html`.

[21] Jim Waldo. Jini Architecture Overview. Available at `http://java.sun.com/products/jini/whitepapers`.

[22] Edward Wobber, Martin Abadi, Michael Burrows, and Butler Lampson. Authentication in the Taos Operating System. *ACM Transactions on Computer Systems*, 12(1):3–32, Feb 1994.

Cha-Cha: A System for Organizing Intranet Search Results

Michael Chen* Marti Hearst† Jason Hong* James Lin*

*Computer Science Department
445 Soda Hall
University of California, Berkeley
Berkeley, CA 94720-1776
{mikechen,jasonh,jimlin}@cs.berkeley.edu
http://www.cs.berkeley.edu/~mikechen

†School of Information Management & Systems
102 South Hall
University of California, Berkeley
Berkeley, CA 94720-4600
hearst@sims.berkeley.edu
http://www.sims.berkeley.edu/~hearst

Abstract

Although search over World Wide Web pages has recently received much academic and commercial attention, surprisingly little research has been done on how to search the web pages within large, diverse *intranets*. Intranets contain the information associated with the internal workings of an organization.

A standard search engine retrieves web pages that fall within a widely diverse range of information contexts, but presents these results uniformly, in a ranked list. As an alternative, the Cha-Cha system organizes web search results in such a way as to reflect the underlying structure of the intranet. In our approach, an "outline" or "table of contents" is created by first recording the shortest paths in hyperlinks from root pages to every page within the web intranet. After the user issues a query, these shortest paths are dynamically combined to form a hierarchical outline of the context in which the search results occur. The system is designed to be helpful for users with a wide range of computer skills. Preliminary user study and survey results suggest that some users find the resulting structure more helpful than the standard retrieval results display for intranet search.

1 INTRODUCTION

Although search over World Wide Web pages has recently received much academic and commercial attention, surprisingly little research has been done on how to search the web pages within large, diverse *intranets*. Intranets contain the information associated with the internal workings of an organization.

Most web site search engines present search results as a ranked list of titles and metadata such as URLs and file size. The context in which the pages exist, and their relationships to one another, cannot be discerned from such a display. Figure 1 shows an example of a list view returned as a result of a query on "earthquake" using a commercial search engine within our university's web site. From this view, it is difficult to tell what the relationships are among the different search hits.

As a remedy, many authors have called for an organization to be imposed on the Web. Directory services such as Yahoo organize web pages according to pre-defined topics. Although such topics are often intuitive, this approach does not scale well because the web pages are assigned to categories manually. Furthermore, most web directories only cover organizations' home pages; standard search engines must be used if the user wants to find information within an intranet.

By contrast, we are interested in organizing the hits returned as the result of queries *within* large, heterogeneous intranets, such as those found at universities, corporations, and other large institutions. These sites are often quite complex, consisting of a wide variety of document genres, including home pages, product information, policy statements, research reports, current events, and news.

Grouping of retrieved documents may be especially important when the user has issued a very short or vague query (user queries tend to consist of only a few words [19, 7]). Short queries often return a heterogeneous set of documents that cover a wide

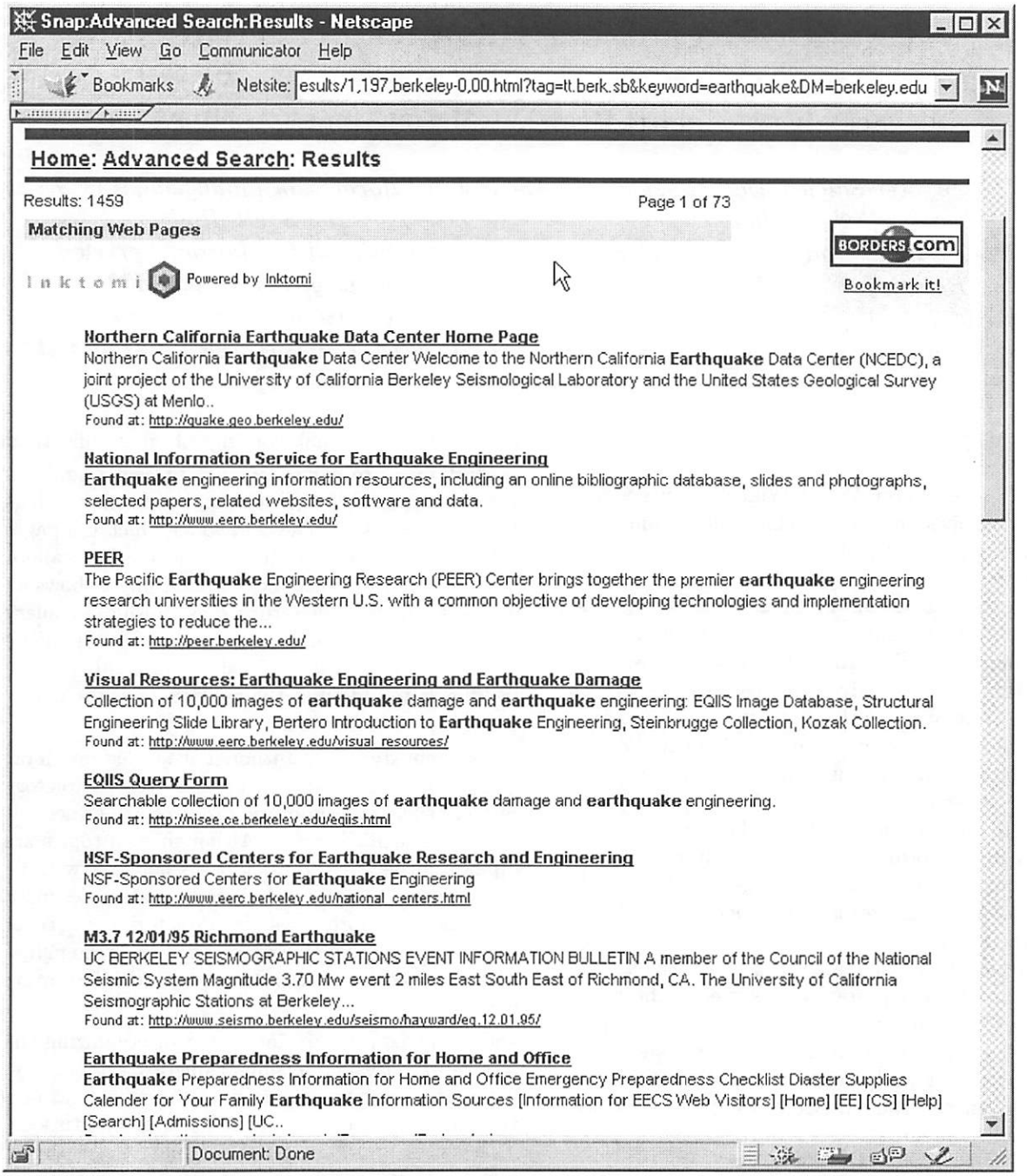


Figure 1: A list view of a commercial search engine on the query "earthquake" within the UC Berkeley intranet.

range of topics. In this situation, retrieval results should suggest the kinds of information available and guide the user to appropriate starting points (such as servers within an intranet).

We have developed a system called Cha-Cha whose goal is to provide fast, well-organized search results for queries within web intranets.[1] Cha-Cha imposes an organization on web site search results by recording the shortest paths, in terms of hyperlinks, from server root pages to every web page within the intranet. After the user issues a query, these shortest paths are dynamically combined to form a hierarchical outline of the context in which the search results occur. This outline structure shows the home pages of the servers on which the search hits occur, as well as the titles of the hyperlinks between the home pages and the search hit.

Figure 2 shows an example on the query "earthquake" within our university's web pages. Note that the interface tightly couples the specification of search with navigation of available hyperlinks.

The top levels of the hierarchy typically identify the servers that the search hits reside on. In this figure, the top levels visible are those associated with a national earthquake center (located on campus), an educational unit on earthquakes (located with a science education site, also on campus), and earthquake research within the campus civil engineering department. The organization pulls out the home pages associated with the search hits, providing not only context in which to embed the hits, but also conveniently indicating the higher level starting points for each subcollection of documents.

We designed the system to make the interface useable by all members of the community, even those with slow computers, low bandwidth connections, and old versions of web browsers. For this reason we use standard HTML and we keep the number of graphics low [26]. We also wanted the interface to look as familiar as possible while still providing added functionality, given research results showing that users do not like to switch to unfamiliar interfaces [21]. We have found that the outline metaphor is familiar enough that users understand the interface rapidly.

The functional goals of the system are to (1) help users better understand the scope of their search results, (2) find useful information more quickly than with a standard view, and (3) learn about the structure of the web site. A major assumption behind this research is that the shortest path of links from a root page to a target web page is meaningful to users. Our subjective experience with the use of this display, and especially in comparison with interfaces that show no context, is that the shortest path information is indeed a useful, coherent way to group the search results. This result is less surprising when taking into account the fact that most hyperlinks are created deliberately by human authors of web pages, who often organize pages into topics.

The remainder of this paper discusses related work, details about the system implementation, and user assessments of the interface.

[1]The name Cha-Cha stands for Contextualized Hierarchical information Access by Chen, Hearst, and Associates. The system can be found at http://cha-cha.berkeley.edu

2 RELATED WORK

This work is closely related to that of SuperBook [9], which demonstrated that showing hit search results in the context of the chapters and sections of the manual from which they are drawn can improve users' information access experiences. The AMIT system [30] indexed a web site covering a specific topic (sailing) in a similar manner, providing a Superbook-like focus-plus-context environment to place search results in context. To our knowledge this system has not been evaluated nor extended to a large heterogeneous web site. The WebTOC system [25] imposes static hierarchical table of contents over a single web site, but focuses on showing the number of pages within a subdirectory and does not integrate the results of search into the outline view. The examples explored for both AMIT and WebTOC were well-structured single web site prototypes organized around a topic (sailing and museum exhibits, respectively).

Cha-Cha demonstrates the application of the idea across a very large, heterogeneous web site that is an order of magnitude larger than those used by these other systems, and is used operationally by thousands of users. Many difficult system-level problems had to be solved to make this approach scale, both because of the larger size of the dataset and the greater heterogeneity in the search results. Cha-Cha also resolves graph-merging issues that these other systems do not address.

A major problem with WebTOC, along with other attempts to provide categorical access to information (such as Yahoo), is that they do not couple navigation with *ad hoc* search (see [1, 12] for more discussion of this point). A navigate-only interface to a large text collection forces the user to contend with the entire contents of the site at once.

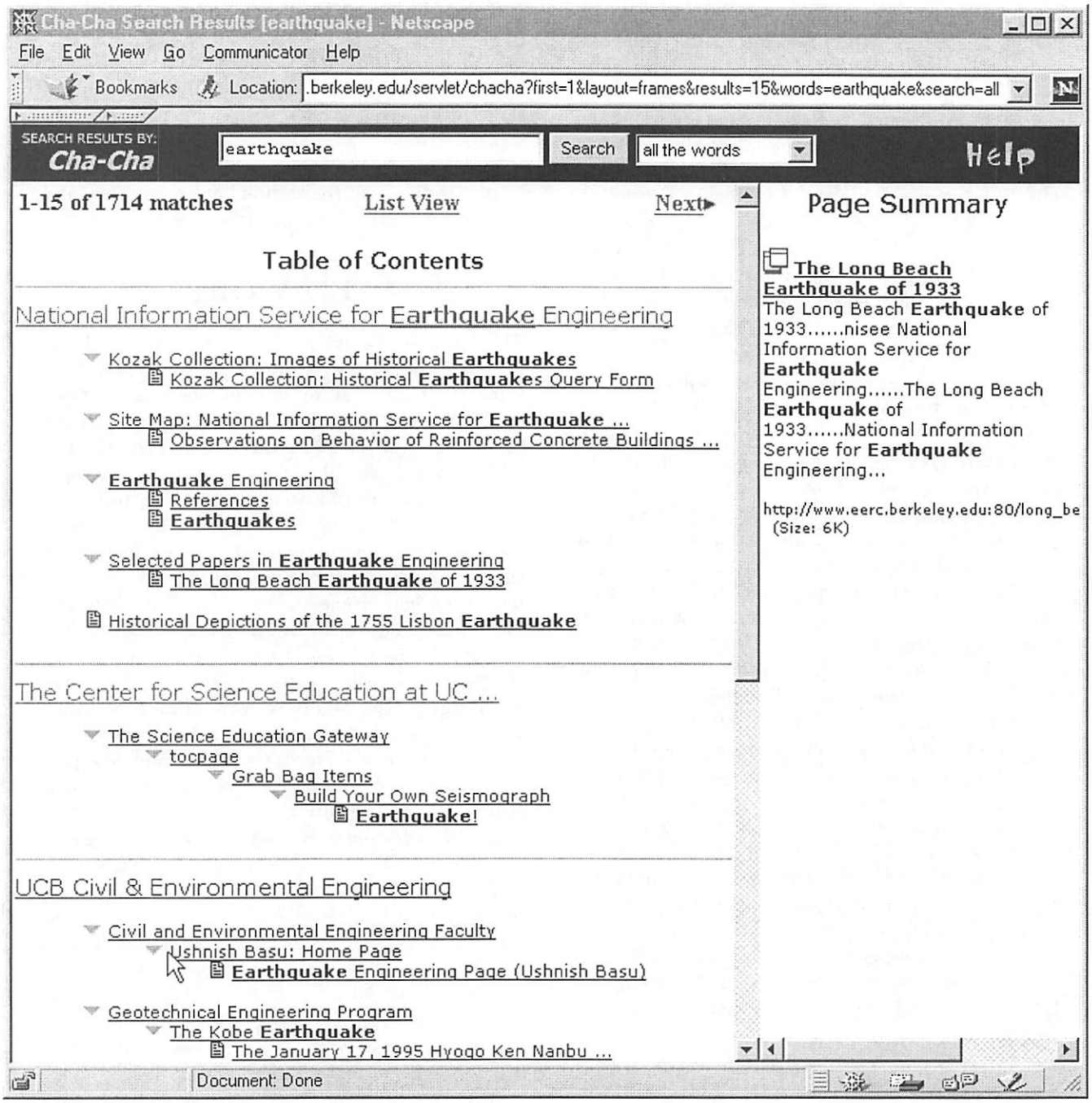


Figure 2: The outline view of the current implementation of Cha-Cha search on the query "earthquake" .

Many web sites show site maps which suffer from the same limitation. Users can browse or navigate the structure of the site, but cannot search within or across that structure.

The WebGlimpse system [22] provides crawling and indexing of an intranet (both local and external links). It also allows system administrators to define link "neighborhoods" in one of two ways: all pages within k links of a given page, or all pages within a file system subdirectory. (We explored this approach in Cha-Cha but found hyperlink path context to be more intuitive than subdirectory paths.) During search, neighborhood information in WebGlimpse is used only to restrict the set of pages that a search is conducted on; it is not used to show the context of the search results. By contrast, Cha-Cha allows search over many different neighborhoods simultaneously, showing in the search results a summary of which neighborhoods the search hits fall into.

The Connectivity Server [3] determines all inlink and outlink information for a given URL across the entire Web. This can be a useful component for a system that ranks pages according to "popularity" as suggested by inlink information (as in Kleinberg et al.'s "authority pages" [15] or the Google system [4]). The authors envision this as a method for viewing the Internet "neighborhood" of a given web page, but do not use the link information to provide context for search results. Furthermore, because results are shown in terms of their relationship to the entire Web, context within a given intranet is not provided.

The WebCutter system [20] (now called Mapuccino) allows the user to issue a query on a particular web site. The system crawls the site in real time, checking each encountered page for relevance to the query. When a relevant page is found, the weights on that page's outlinks are increased. The subset of the web site that has been crawled is depicted graphically in a nodes-and-links view. Other researchers have also investigated spreading activation among hypertext links as a way to guide an information retrieval system [10, 24].

The emphasis of these systems is on the dynamic crawling of the web sites. Unfortunately, this kind of display does not provide the user with information about what the *contents* of the pages are, but rather only shows their link structure. This is in contrast with Cha-Cha which retrieves all search hits at once, whether or not they are close to a starting point. Cha-Cha also emphasizes the display of the contents in a readable form, as previous research indicates that users find this more helpful than nodes-and-links views [14, 28].

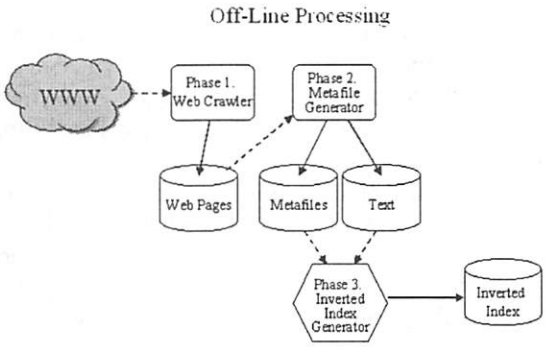


Figure 3: The architecture for the off-line indexing components of Cha-Cha.

3 SYSTEM IMPLEMENTATION

The Cha-Cha system has two main parts: an off-line indexing component in which the web site is crawled and the metadata and indexes are generated, and an on-line query processing component in which the system receives and responds to user requests. Figures 3 and 4 illustrate the two main components of the architecture; all code is written in Java unless otherwise noted.

The indexing component has three main phases: the web crawler, the metafile generator, and the indexer (see Figure 3). The web crawler stores a mirror of the intranet's web pages on its local file system. The metafile generator processes these files to precompute shortest path information and other meta-information, storing the metadata on the local file system. The indexer converts the metadata and the text into an inverted index to be used by the search engine backend, the Cheshire II system [18]. Each of these components is discussed in detail below, followed by a discussion of the query processing component.

3.1 The Web Crawler

In order to ensure thorough web coverage and to create the necessary metadata, we have written a custom web crawler. To uniquely identify each site, we record the MD5 hash[2] of the root page of each

[2] http://www.ietf.org/rfc/rfc1321.txt

host name found. If the root page of a new host has the same hash as a previous host, then the new host name is mapped to the previously known name in all URLs subsequently encountered [8]. This technique eliminates duplicates caused by host name aliases, DNS round-robin, virtual hosts, and unique servers that mount the same web pages at the root level. A global breadth-first search algorithm is used to order the URLs [5], and a page is retried for up to three times if errors are encountered during retrieval. Dynamically generated pages are not crawled.

The web crawler is given a list of URLs from which to start (e.g., the home page at www.berkeley.edu). The crawler is restricted to following links that fall only within a set of domains (e.g., all of *.berkeley.edu), while obeying the robots exclusion standard.[3] The crawler mirrors the full text of the web pages onto disk. This is needed to allow for extraction of sentences for page summaries.

3.2 The Metafile Generator

After the web pages are recorded, the metafile generator extracts hyperlink relationships. It begins at the root page of the main server. The HTML of the current page is parsed and all the outlinks to pages that have not yet been processed are placed on a queue. The system stores information about the page in a disk-based storage system.[4] This information includes a count of the shortest distance found so far from the root to the page, along with the corresponding shortest path(s). Then the next page is taken off the top of the queue and the process repeats until the queue is empty. If later in this process a page is encountered again, and was reached via a shorter path than before, the database entry for the page is updated to reflect the shorter path. Pages are allowed to contain multiple shortest paths (of equal length). Simple algorithms are used to assign categories such as personal home page and department home page, which can later be used as a search criteria. The metafile generator also records the title, domain, URL, page length, date last modified (if available), inlinks and outlinks.

In the initial implementation of the system, the root node was the home page of our institution, and all shortest paths were generated relative to this root page. However, this approach can produce misleading results, because sometimes the shortest path within a local subdomain is more meaningful than the shortest path computed globally or the entire intranet. (A related idea is discussed by Terveen and Hill [29].) To remedy this problem, we have implemented a variation of the algorithm that combines *local* and *global* shortest path information. This allows the shortest path information to be organized more modularly.

To make use of both local and global information, a two-pass scheme is used for metafile generation. First, shortest paths are computed within each logical domain, starting from the root page associated within the local logical domain. Then, global shortest paths are computed in the same way, beginning with the home page of the entire organization. Whenever a page is found using the global pass that has not yet been encountered in any of the local passes, this page and its shortest paths are added to the metafile database. This second pass finds "orphan" pages that might have been missed by the local passes, but gives priority to shortest paths found locally.

The two-phase approach trades file reading time for simplicity, as files have to be read and parsed twice. This inefficiency can be removed by keeping track of both global and local paths simultaneously for each page. The time to generate the metafiles for a 200K collection is currently less than 5 hours, which is adequate for our institution.

3.3 The Indexer

The search engine backend is the Cheshire II system [18], (written in C). This system creates an inverted index [2] based on the full text of the pages and attribute information found in the metadata files. Cheshire II works with SGML markup. After reading in a Document Type Definition (DTD) [27] describing the format of the metafiles, the system creates indexes that provide efficient access to search terms embedded within the full text, titles, and other forms of metadata.

3.4 Query Processing and the User Interface

Because studies show that many users are reluctant to switch from familiar to unfamiliar interfaces and systems, one of our principle design goals to was to create an interface as similar as possible to the status quo, while providing added functionality.

Users access the system via a web browser that communicates with the Cha-Cha frontend, which is

[3] http://info.webcrawler.com/mak/projects/robots/norobots.html

[4] Written in C, see http://www.sleepycat.com.

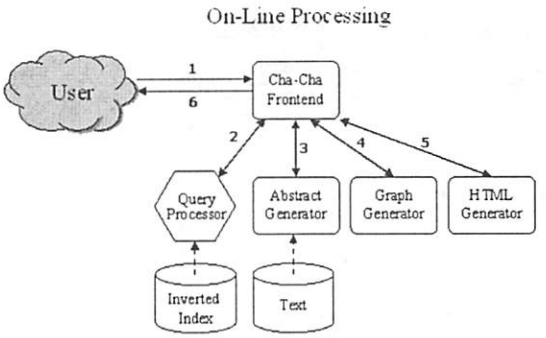


Figure 4: The architecture for the on-line (interactive) query processing components of Cha-Cha.

a Java servlet hosted on an Apache server.[5] (See Figure 4, step 1.) The Cha-Cha frontend formats the user query and sends it to the Cheshire II backend (step 2). The frontend requests the metadata for the first k hits, where currently k is set to 25. The output of queries are ranked using a probabilistic algorithm [6]. The two-tier design will allow us in future to use load-balancing techniques across several servers so the system can scale well.

The user can place the interface into one of two modes: list mode or outline mode. If list mode is selected, the system creates an HTML page containing the titles, summaries, page size, date, and URL for the first k hits (step 3). The page also shows the total number of hits for the query. If there were more than k hits for the query, hyperlinks are shown at the top and bottom of the page indicating that more pages of search hits are available. Titles are hyperlinked to the actual pages to which they refer.

If outline mode is selected, the system builds up a hierarchy from the shortest paths associated with each of the k hits (step 4; how this is done is described below). The outline layout is placed into two frames; the first acts as a "table of contents" or a "category hierarchy." This outline view is generated by recursively traversing the hierarchy data structure and generating HTML (step 5). Small icons are associated with the titles of search hits. These icons, if clicked, bring up a display of the document summary in the righthand frame. The titles within the table of contents (search hits as well as their contextualizing information) are hyperlinked to the actual web pages to which they refer. Finally, the HTML pages are displayed in the user's client browser (step 6).

[5] http://www.apache.org

3.5 The Page Summaries

There is strong evidence that highlighting query terms in the context in which they occur in the page helps user determine why the document was retrieved and how it might be relevant to their information need [17, 23].

The keyword-in-context (KWIC) summaries consist of a sentence extracted from the beginning of the document followed by up to three sentences containing search terms. The search terms themselves are shown in boldface. Figure 5 shows an example on the query "contact lens". If the query consists of q different terms, sentences containing all q are favored over those with fewer, failing these, sentences containing $q-1$ are favored, etc. In the case of ties, sentences from the beginning of the document are favored over those found later. To help retain coherence of the excerpts, selected sentences are always shown in order of their occurrence in the original document, independent of how many search terms they contain.

3.6 The Graph Merging Algorithm

After all of the shortest paths have been found, they are used to build a hierarchy, starting from the root, in which common paths are merged together. This merging is partly responsible for the grouping and compression of search results.

As noted above, there is often more than one shortest path to any particular search hit. This means that a graph built from all of the shortest paths is a directed acyclic graph (DAG). We choose to show search hits only once within the hierarchy in order to reduce the amount of extra information presented to users, and because we suspect that seeing the same page in two locations in the hierarchy will be more confusing than helpful. We also assume that it is better to group many related hits together within one subhierarchy, if possible, rather than scattering the same set of hits across many different subhierarchies. This assumption is based on the results of our empirical work which showed that documents relevant to a query tend to fall into the same one or two clusters when text clustering is employed [13]. In general, grouping related documents together seems to facilitate rapid discarding of non-relevant groups [11].

Although we do not use clustering for computing similarity here, we assume that a human author groups pages together via hyperlinks because they are similar to one another in some sense. Thus hyperlink structure can provide a kind of supervised

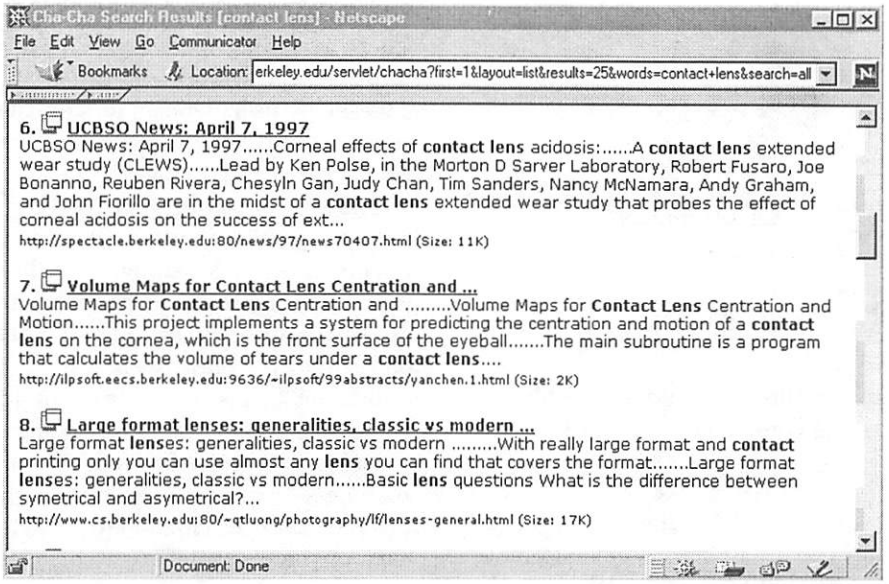


Figure 5: Examples of keyword-in-context summaries.

similarity metric that document clustering attempts to uncover in an unsupervised manner. The hyperlink structure has the added advantage of human-understandable labels (in the form of the page titles) and a uniform granularity of detail, both of which are lacking in clustering algorithms [11].

Based on these assumptions, the layout has the following goals (assuming that search hits are leaves in the final hierarchy).

(i) Group (recursively) as many pages together within a subhierarchy as possible; avoid (recursively) branches that terminate in only one hit (leaf).

(ii) Remove as many internal nodes as possible while still retaining at least one path to every leaf.

(iii) Remove as many edges as possible while still retaining at least one path to every leaf.

Items (ii) and (iii) are based on the assumption that the fewer levels of the hierarchy shown the better, since pilot studies suggest users prefer to have less extra information rather than more.

These desiderata require a non-standard graph algorithm[6] whose goal is to find the smallest subset of nodes that covers all the nodes one level below.[7] To do this correctly, every possible subset of nodes at depth D should be considered to determine the minimal subset which covers all the nodes at depth $D+1$. However, this would require 2^k checks if there are k nodes at depth D. Instead, a heuristic approach is used.

The main idea is to order nodes at each depth according to how many children they have and *eliminate* those nodes that do not uniquely cover nodes at the depth below them.

First, a top-down pass determines the depth of each node and the number of children it links to one level below. Next a bottom-up pass works from the deepest nodes (the leaves) up to the root. In other words, say the current deepest level is $D+1$. The nodes at level D are sorted in ascending order according to how many active children they link to at depth $D+1$. A node is *active* if it has not been eliminated in a previous step.

Every non-leaf node at level D is a *candidate* for elimination from the final graph. Those candidates with the least number of children are examined for potential to be eliminated first, because of goal (i). For each candidate C, if C links to one or more ac-

[6] Minimum spanning tree is inappropriate because internal nodes, as well as edges, are to be eliminated. A depth-first traversal in which the counts of the leaves are propagated up does not work because the graph is a DAG: If node N has two internal node children $N1$ and $N2$ and they both point to leaf L, N will be assigned an erroneous count of two children in such a traversal.

[7] To obtain a truly optimal result the algorithm should optimize elimination of nodes and edges at all levels of the hierarchy, not just between one level and the next. There is at least one case in which the heuristic approach using only one level at a time yields a suboptimal result, but in practice this kind of situation seems to occur only rarely.

tive nodes at depth $D+1$ that are not covered by any of the other active candidates, then C cannot be eliminated. Otherwise C is removed from the active nodes list for depth D. After a level is complete, there are no active nodes at depth D that cover only nodes that are also covered by another active node at D. This procedure continues recursively up to the root. It works despite the DAG structure because the edges used correspond only to shortest paths at every level. The running time is order $(k\ log\ k) + 2k$ for each level, instead of 2^k. Figure 6 shows the pseudocode for the main loops of the algorithm.

```
mergeShortestPaths(paths) {
  init();
  NodesAtDepth[] =
     findShortestPathDepths(paths);
  pruneNodes(NodesAtDepth);
  return(buildPathTreeFromNodes());
}

pruneNodes(NodesAtDepth[]) {
  Vector tobeCov, candidates;
  for(int d = MaxDepth-1; d>0; d--) {
     tobeCov = NodesAtDepth[d];
     candidates = NodesAtDepth[d-1];
     NodesAtDepth[d-1] =
      getCovering(tobeCov, candidates);
  }
}
```

Figure 6: Pseudocode for the main loops of the graph merging algorithm.

After nodes have been eliminated, the hierarchy must be built up while still attempting to retain the rank ordering from the search engine. This is accomplished as follows. The leaf nodes (search hits) are sorted in ascending order according to their rank in the search results (a rank of 1 means the best-ranked hit). Beginning with an empty tree and the first hit, a path is found from that hit through the active nodes at each level above it to the root. The path must travel through edges from the original set of shortest paths. The parent with the largest number of still active children is chosen, in order to help achieve goal (i). When the root is reached, a new path has been created; this path becomes the beginning of the output tree. This procedure is repeated with the rest of the search hits in ranked order, with an added check: when selecting a parent, if one of the parent choices is already in the final tree, chose that over other parent choices (to help achieve goal (ii)).

The tree is converted to an HTML hierarchy by traversing it in a depth-first manner. The choice of which sibling to traverse next is determined by the order in which the siblings were entered into the tree. Thus the rank ordering is preserved as much as possible while still grouping search hits together within subhierarchies.

3.7 System Status

Cha-Cha has been available from our institution's home page since August 1998, receiving on average approximately 3000 queries per weekday during the school year. During a one week period, for 17831 queries, the outline view was used 16006 times for 14330 unique IP addresses. (Since the outline view is the default setting we cannot assume that people deliberately choose this view.) Our target response time was 3 seconds per query on average. We achieved this goal; for 17831 queries run during one week in August, the average time required by the system was 3.02 seconds running on a Sparc Ultra II. The search engine backend took 2 seconds on average while the Java frontend took 1 second. More recently we have obtained a Sun Enterprise 450. For 65440 queries during the month of July, 1999, the average time was 2.4 seconds/query. The current index covers more than 200,000 web pages and several groups within our institution are using Cha-Cha to search over their site's pages. All that is needed to enable this facility is the customization of a short HTML form.

4 USER ASSESSMENTS

As mentioned above, the functional goals of the system are to help users better understand the scope of their search results, find useful information more quickly than with a standard view, and learn about the structure of the web site. These are all difficult to evaluate empirically [16]. Below we describe a pilot study that attempts to assess some of these factors, a follow-up user study, and the results of a survey intended to uncover user preference information.

4.1 A Pilot Study

To assess the relative merits of the system we conducted a pilot study followed by a full study. The

	Question	Yes	No	NA
1.	I often find the outline view helpful.	56	31	75
2.	I often find the outline view confusing.	37	49	76
3.	The outline view sometimes helps me find information that a standard search engine does not.	57	30	75
4.	The outline view introduces unnecessary clutter.	37	51	78
5.	The outline view would be better if it showed less information.	33	51	78
6.	I usually prefer the list view with the summaries over the outline view.	42	39	81
7.	I usually prefer the Cha-Cha outline view to standard search engine results listings.	35	44	83

Table 1: Responses of 162 survey participants on questions about Cha-Cha in particular.

pilot study used an earlier version of the system on a smaller data set (about 10,000 pages). Seven people participated and four versions of the interface were compared. The participants had not used Cha-Cha prior to the study, and had no training on the interfaces.

The results showed that participants were able to understand a version of the outline view and found it easy to use. When participants were timed on eight question-answering tasks, the average time for outline view was 72.4 seconds/query while for the list view it was 99.7 seconds/query. Due to the small nature of the study, these results are only suggestive.

4.2 Follow-up Study

We conducted a more extensive user study on a later version of the system with a larger number of pages (about 100,000), but with inferior path structure, because the local/global distinction had not yet been implemented. This study involved 18 participants, 9 males and 9 females, from a wide range of undergraduate majors (one participant was a graduate student). The study compared only the outline view and the enhanced KWIC list view. This version of the list view contains more information than a standard web search engine, showing an abstract containing keywords in the context in which they occur in the web page, rather than just the first one or two sentences of the web page.

Study participants scored the two views on the question "How often would you use this interface if it were available" on a scale from 1 (never) to 7 (often). The outline view received an average score of 4.6 while the list view received a score of 4.9 (differences were not significant). However, when participants were timed on eight question-answering tasks, the average time for outline view was 109 seconds/query while for the list view it was 120 seconds/query (again, differences were not significant).

4.3 Survey Results

To attempt to measure preference information, a survey was recently placed on the home page for our institution, inviting the user community to express opinions about the usability of Cha-Cha and a commercial search engine, both of which provide search over the UC Berkeley web pages.

Because those people who choose to answer the survey are somewhat self-selected, the results should be considered to come from a biased sample. Nevertheless, the results should provide useful ballpark estimates on perceived usability and user preference. After several weeks, 162 responses were gathered. 96 responses were from UCB undergraduates (apparently; this was the default choice), 23 were from UCB Staff, 10 were UCB graduate students, with the remaining 33 from other categories.

Respondents were asked if they had to choose between Cha-Cha and the commercial search engine, which would they prefer. 38 respondents preferred Cha-Cha, 21 preferred the commercial engine, 61 marked no opinion, and 42 made no choice at all. Thus, of those who expressed an opinion of one system over the other, 64% preferred Cha-Cha. Bearing in mind that most users are reluctant to adopt to new interfaces, this is an encouraging statement in favor of this approach.

Table 1 shows the questions pertinent only to Cha-Cha and the number of responses. For these questions, respondents could mark Yes, No, or mark neither choice (NA). Over half the respondents find the outline view helpful and claim to be able to find information using it that they can't otherwise find.[8]

[8]The last question seems to be worded in a confusing manner, because in many cases respondents who were positive about other aspects of Cha-Cha responded No to this question.

If only those respondents who expressed an opinion are taken into account, then about two thirds find the outline view helpful ($56/87 = .64$).

Anecdotally, users that tell us they like Cha-Cha tell us it is often useful for especially hard-to-find information. In these circumstances they often end up looking at the hyperlink one or two levels above a hit and explore the web site in that general area.

We think there are three main reasons between the discrepancies in the preference results we see between the follow-up user study and the survey. First, the local/global improvements to the path generation discussed in Section 3.2 were made after the user study but prior to the survey. We think these changes improve the meaningfulness of the hierarchies in many cases. Second, we suspect that users grow to prefer the outline view as they become more familiar with the system. The benefits of a new interface cannot always be assimilated immediately, and may be especially difficult to appreciate in a timed test like that of our follow-up study. Survey users used the interface to achieve their own ends at their own pace. Third, some users may prefer Cha-Cha over the commercial search engine because Cha-Cha's list view is richer than that of the commercial search engine, and because the commercial engine displays advertisements.

5 CONCLUSIONS AND FUTURE WORK

We have described the motivation, architecture, algorithms, and user assessment results for a search engine interface that organizes search results over large intranets into coherent structures.

We are encouraged by the user study results, the initial survey results, and informal reactions by users of the system. We argue that if we are providing an interface that is preferred by a substantial subset of the user population, then we are providing a useful service. During the course of the pilot study and the fuller study, we have learned about improvements participants would like to see in the system design that might make the outline view substantially more effective than standard views. In future, we plan to investigate how to incorporate semantic information into the interface. We also plan to index a wide variety of organizations' intranets.

Acknowledgements We thank Ray Larson for adding code to Cheshire II to make it suit our needs, Keith Bostic and Margo Seltzer of SleepyCat Software Inc for allowing use of their database software package, Eric Brewer for arranging access to Inktomi for an earlier version of the system, Kevin Heard and Bryan Lewis for technical assistance, and Hal Varian for general support. Additionally, we thank Sun Microsystems for a generous equipment donation.

References

[1] M. Agosti, G. Gradenigo, and P.G. Marchetti. A hypertext environment for interacting with large textual databases. *Information Processing & Management*, 28(3):371–387, 1992.

[2] Ricardo Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval.* Addison-Wesley Longman Publishing Company, 1999.

[3] Krishna Bharat, Andrei Broder, Monika Henzinger, Puneet Kumar, and Suresh Venkatasubramanian. The connectivity server: fast access to linkage information on the web. In *Proceedings of the Seventh International World Wide Web Conference (WWW 7)*, 1998.

[4] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. In *Proceedings of the Seventh International World Wide Web Conference*, 1998.

[5] Junghoo Cho, Hector Garcia-Molina, and Larry Page. Efficient web crawling through URL ordering. In *Proceedings of the Seventh International World Wide Web Conference (WWW 7)*, 1998.

[6] William S. Cooper, Fredric C. Gey, and Aitoa Chen. Probabilistic retrieval in the TIPSTER collections: An application of staged logistic regression. In Donna Harman, editor, *Proceedings of the Second Text Retrieval Conference TREC-2*, pages 57–66. National Institute of Standards and Technology Special Publication 500-215, 1994.

[7] W. Bruce Croft, Robert Cook, and Dean Wilder. Providing government information on the internet: Experiences with THOMAS. In *Proceedings of Digital Libraries '95*, pages 19–24, Austin, TX, June 1995.

[8] F. Douglis, A. Feldmann, B. Krishnamurthy, and J. Mogul. Rate of change and other metrics: a live study of the world-wide web. In *Proceedings of the 1997 Usenix Symposium on Internet Technologies and Systems (USITS-97)*, Dec 1997.

[9] Dennis E. Egan, Joel R. Remde, Thomas K. Landauer, Carol C. Lochbaum, and Louis M. Gomez. Behavioral evaluation and analysis of a hypertext browser. In *Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems*, pages 205–210, May 1989.

[10] H. P. Frei and D. Stieger. The use of semantic links in hypertext information retrieval. *Information Processing & Management*, 31(1):1–13, 1994.

[11] Marti A. Hearst. The use of categories and clusters in organizing retrieval results. In Tomek Strzalkowski, editor, *Natural Language Information Retrieval*, pages 333–374. Kluwer Academic Publishers, 1999.

[12] Marti A. Hearst and Chandu Karadi. Cat-a-cone: An interactive interface for specifying searches and viewing retrieval results using a large category hierarchy. In *Proceedings of the 20th Annual International ACM/SIGIR Conference*, Philadelphia, PA, 1997.

[13] Marti A. Hearst and Jan O. Pedersen. Reexamining the cluster hypothesis: Scatter/gather on retrieval results. In *Proceedings of the 19th Annual International ACM/SIGIR Conference*, pages 76–84, Zurich, Switzerland, 1996.

[14] Adrienee J. Kleiboemer, Manette B. Lazear, and Jan O. Pedersen. Tailoring a retrieval system for naive users. In *Proceedings of the Fifth Annual Symposium on Document Analysis and Information Retrieval (SDAIR)*, Las Vegas, NV, 1996.

[15] Jon Kleinberg. Authoritative sources in a hyperlinked environment. In *Proceedings of the 9th ACM-SIAM Symposium on Discrete Algorithms*, 1998.

[16] Eric Lagergren and Paul Over. Comparing interactive information retrieval systems across sites: The trec-6 interactive track matrix experiment. In *Proceedings of the 21st Annual International ACM/SIGIR Conference*, pages 164–172, 1998.

[17] Thomas K. Landauer, Dennis E. Egan, Joel R. Remde, Michael Lesk, Carol C. Lochbaum, and Daniel Ketchum. Enhancing the usability of text through computer delivery and formative evaluation: the superbook project. In C. McKnight, A. Dillon, and J. Richardson, editors, *Hypertext: A Psychological Perspective*, pages 71–136. Ellis Horwood, 1993.

[18] Ray R. Larson, Ralph Moon, Jerome McDonough, Lucy Kuntz, and Paul O'Leary. Cheshire ii: Design a next-generation online catalog. *Journal of the American Society for Information Science*, 47(7):555–567, 1996.

[19] X. Allan Lu and Robert B. Keefer. Query expansion/reduction and its impact on retrieval effectiveness. In Donna Harman, editor, *Proceedings of the Third Text Retrieval Conference TREC-3*, pages 231–239. National Institute of Standards and Technology Special Publication 500-225, 1995.

[20] Y. S. Maarek, M. Jacovi, M. Shtalhaim, S. Ur, D. Zernik, and I. Z. Ben Shaul. WebCutter: A system for dynamic and tailorable site mapping. In *Proceedings of the Sixth International World Wide Web Conference*, pages 713–722, 1997.

[21] Robert R. Mackie and C. Dennis Wylie. Factors influencing acceptance of computer-based innovations. In Martin Helander, editor, *Handbook of Human-Computer Interaction*, pages 1081–1106. Springer Verlag, 1988.

[22] Udi Manber, Mike Smith, and Burra Gopal. WebGlimpse – combining browsing and searching. In *Proceedings of 1997 Usenix Technical Conference*, 1997.

[23] Gary Marchionini. *Information Seeking in Electronic Environments*. Cambridge University Press, 1995.

[24] Filippo Menczer and Richard K. Belew. Adaptive information agents in distributed textual environments. In *Proceedings of the 2nd International Conference on Autonomous Agents (AGENTS-98)*, pages 157–164, May 1998.

[25] A. Nation. Visualizing websites using a hierarchical table of contents browser: Webtoc. In *Proceedings of the Third Conference on Human Factors and the Web*, Denver, CO, 1997.

[26] Jakob Nielsen. *Designing Excellent Websites: Secrets of an Information Architect*. New Riders Publishing, 1999. To appear. See www.useit.com.

[27] Natanya Pitts-Moultis and Cheryl Kirk. *XML Black Book*. The Coriolis Group, 1999.

[28] Marc Sebrechts, Joanna Vasilakis, Michael S. Miller, John V. Cugini, and Sharon J. Laskowski. Visualization of search results: A comparative evaluation of text, 2d, and 3d interfaces. In Marti A. Hearst, Fredric Gey, and Richard Tong, editors, *Proceedings of the 22nd Annual International ACM/SIGIR Conference*, pages 3–10, Berkeley, CA, 1999.

[29] Loren Terveen and Will Hill. Finding and visualizing inter-site clan graphs. In *Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems (CHI '98)*, Los Angeles, CA, April 1998. ACM.

[30] Kent Wittenburg and Eric Sigman. Integration of browsing, searching, and filtering in an applet for web information access. In *Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems, Late Breaking Track*. ACM, 1997.

A Document-based Framework for Internet Application Control

Todd D. Hodes and Randy H. Katz
Computer Science Division
University of California, Berkeley
Berkeley, CA 94720
{hodes,randy}@cs.berkeley.edu

Abstract

This paper motivates and details a document-based framework for manipulating the components that comprise distributed Internet applications. In the framework, XML documents are used to describe both server-side functionality and the mapping between a client's applications and the servers it accesses. Our system model contrasts with explicitly context-aware application designs, where location information must be explicitly manipulated by the application to affect change; instead, a middleware layer is interposed between client applications and services so that invocations between the two can be transparently remapped. This approach is useful for a subset of application domains, including our example domain of "remote control" of local resources (e.g., lights, stereo components, etc.). We illustrate how the framework allows for 1) remapping of a portion of an existing user interface to a new service, 2) viewing of arbitrary subsets and combinations of the available functionality, and 3) mixing dynamically-generated user interfaces with existing user interfaces.

The use of a document-based framework in addition to a conventional object-oriented programming language provides a number of key features. One of the most useful is that it exposes the mappings between programs/UIs and the objects to which they refer, thereby providing a standard location for manipulation of this indirection.

1 Introduction

Many university and industry groups have projects investigating compositional frameworks for large-scale distributed systems; examples include CalTech's InfoSpheres [11], MIT's Oxygen [3], UCB's Ninja [18], IBM Almaden's TSpaces [31], HP's e-Speak [8], Sun's Jini/EJB atop Java [28, 26], and OMG's CORBA [19]. The basic idea is to enable groups of remote objects on independent Internet hosts to be used together ("federated") to perform tasks via the provision of edifices such as component discovery (e.g., the Jini Lookup service or Ninja SDS [2]) and remote invocation.

Building large-scale software from distributed components, or "services,"[1] is a relatively new area of study, one that has challenges that are inherently different from those in monolithic or client/server program design.

1.1 The Challenge of Orthogonalizing Component Management and Component Usage

Conventional component-based software specifies component locations locally, internal to the application. Keeping such information at the level of the application complicates the process of adapting to components that fail or change due to mobility: the application must either deal with this itself, or a separate component must understand the application-specific configuration files/APIs. To reduce the burden on applications designers and enable generic service management middleware, the component management (locating/spawning/etc.) functionality and component usage functionality can be either partially or completely orthogonalized. This allows the designers to focus on exposing the aggregate functionality to the user, leaving the details of manipulating applications' component references to the middleware. Benefits of such partial or complete orthogonalization of layers is described in [7]. But, the remaining question is, how might we implement such layering in this domain?

[1] We call our framework components "services" to contrast with the more generic term "objects." A service is any entity that can be invoked over the Internet using a known messaging format. Thus, a web server is a "service," as might be a VCR that is connected to the network and advertising its control interface, as in [9].

1.2 The Challenge of Heterogeneous Interfaces

In the context of distributed evolution of components (i.e., by groups scattered across the Internet), there is the potential for independent class hierarchies to be built such that semantically identical objects may either not type-match due to a difference in the interface name, or not type-match because they have differing interfaces. For example, a three-state "on/off/dim" light switch may not type match with a continuous dimming light switch, or a LightSwitchInterface may not match a PowerSwitchInterface. This leads to a situation where the aggregate system is not composed of objects with consistent interfaces and potentially different implementations; instead, it is composed of both heterogeneous interfaces and heterogeneous implementations. The former case is cleanly handled by any of the aforementioned distributed object systems, while the latter is not.

One approach to addressing these problems is to allow application programs to be downloaded on-the-fly to hand-held devices and uploaded to local computers [9]; for example, as Java applets. The difficulty of this approach, though, is that it does not allow the end-user to customize applications for interaction with a heterogeneous set of services as related entities. In other words, it cannot overcome minor differences in protocol — even for functionally identical services — because the applications are opaque. For example, in the Jini [28] model, a discovered service exposes its interface by passing a Java applet to the client. The applet is allowed to – even encouraged – to use an application-specific protocol (atop RMI) between itself and the host server. If light switch controls were designed using such a model, a user would need to download an applet each time he or she encountered a light switch from a different manufacturer or with different options. The end result of this is that, though the functionality is exposed, it is not in a form amenable to manipulation: the client/server protocol may be opaque.

Given an inability to standardize all functional interfaces and the need to avoid using only opaque mobile code, is there an intermediate solution that balances the need to expose interfaces with the need to agree on protocol standards?

1.3 A Solution Framework: Externalize Component State in Documents

This paper proposes that there is a commonality in the two challenges, and that a single framework can support solutions to each. This framework is a middleware layer that sits above any existing distributed object layer [13]. The new layer extends the traditional distributed component programming approach by forcing applications to expose a portion of their system state as documents.

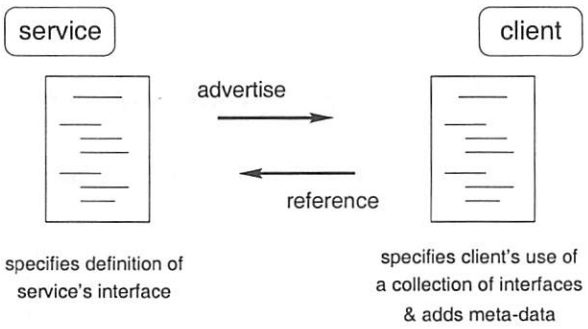


Figure 1: The Document-based Model: services are described by static documents that advertise the definition of their interface; clients maintain documents that indicate how the a collection of interfaces are used.

Specifically, a programs' remote object usage is separated from its internals, in a manner exposing these locations rather than hiding them. This enables composition by allowing this state to be manipulated by editing the component description documents — a function that can be affected by a third-party independent of the original application. We call this novel use of standardized externalization a "document-based" approach. In the framework, application programs and user interface programs are associated with documents that provide either description of the available services or flexible association of user interfaces to these services.

This document-based distributed object management framework is illustrated in Figure 1.

The document-based approach doesn't add any functionality that couldn't otherwise be built into an application, but instead forces there to be a standard layer of indirection between certain references. This indirection is useful in separating the concerns of the applications and the middleware. It exposes the mapping between programs/UIs and the objects to which they refer, thereby providing a standard location for users (and their programs) to manipulate these mappings. Such a system model contrasts with explicitly context-aware application designs [24], where location information is either centralized or must be explicitly manipulated by the application to affect change. Instead, our approach is to interpose a middleware layer between client applications and services so that invocations between the two can be

transparently remapped. This approach is useful for a subset of application domains, including that of "remote control" of local resources (e.g., lights, VCRs, stereo components, etc.), an example we will treat in detail in Section 5.

This disassociation of programs/UIs from the objects they reference is similar to the Model/View/Controller (M/V/C) architecture from Smalltalk [12]. In the M/V/C architecture, data (the model) is separated from the presentation of the data (the view) and events that manipulate the data (the controller). Similarly, documents in our system act as the glue that associates data to user interfaces/programs that manipulate and view that data.

How might this framework address the two challenges presented Sections 1.1 and 1.2: implementing component management outside individual applications, and providing a place where heterogeneity in interfaces can be detected and addressed?

To enable management/usage orthogonality, we can externalize references in the form of a standardized document format used by both the application and middleware. The document is referenced by the application whenever it wishes to make distributed object invocations, and is referenced and/or modified by the middleware to check or update component locations.

To address heterogeneity in interfaces, we can use our document model to externalize component descriptions and user interface mappings. This hybridizes features of the two basic approaches discussed above, allowing downloading/uploading of code fragments (as specified by the documents) while imposing only a standard for interface description and manipulation rather than application-specific interface types. Structural typing, similar to the approach of [25], can used for matching of expected client interfaces to advertised service interfaces rather than named typing. Because our documents are described in a dialect of XML, XML queries [4, 22] can be used for this structural type mapping, and matching on portions of an interface (a form of subtyping) is naturally supported. Specifically, the use of the document-based approach solves two aspects of the heterogeneous component interface problem: standardization of entity descriptions as a step toward interoperable manipulation of such entities, and specification of the mapping between components and user interfaces that can access them. It does not directly solve the final portion of the problem, that of generating entities that wrapper incompatible interfaces. We mention our proposed solution to this final portion of the problem — using intermediaries that provide pairwise mapping of method invocations based on structural typing — in Section 8's discussion of continuing work.

An additional feature of solving the heterogeneous interface problem in this way is that fractions of service descriptions that are not handled by any client program can be used directly to dynamically generate a user interface. This turns out to be quite useful in the domain of "remote control" applications, our area of primary investigation.

The rest of this paper elaborates on our document-based framework. It is structured as follows. Section 2 describes the investigation approach. Section 3 introduces XML, motivates its use, and describes the markup tags we use in the interface description documents. It also explains how these tags are used in communication endpoint resolution and choosing user interfaces. Section 4 gives details on automatic user interface generation. Sections 5–6 give examples of the use of the framework for "remote control" and show document markup examples along with their related applications. It includes examples of

- remapping of a portion of an existing room's user interface to a new room's set of controls (for example, due to movement of the terminal)
- exposing arbitrary subsets and combinations of the functionality available to the user, and
- mixing dynamically-generated user interfaces with existing user interfaces to address the case where a native user interfaces are not available for all the components the user wishes to access.

Section 7 describes related work, Section 8 describes continuing work, and, finally, Section 9 summarizes and concludes.

2 Project Approach

Leveraging the eXtensible Markup Language (XML) [30] for syntax, we develop our document schema as an XML document type definition (DTD). The schema, called ISL, provides markup tags for language-independent service descriptions and for mapping UIs (programs) to referenced services and vice-versa. We then build software that can heuristically generate UIs from these service descriptions without associated custom UIs, and allows mix-and-match use of custom and

generated UIs. Additionally, we built an index application that lists the collection of available UIs and services, allowing a combination of them to be interactively selected for presentation on the user's machine. Finally, we prototype applications that use the model, manually constructing and editing documents to simulate how programs would automatically manipulate them.

Our prototype application is a "universal remote control" based on a set of location-based services [9, 10]. The application provides software remote control of various rooms' devices from a mobile, wirelessly-connected laptop computer. Manipulations of application documents allows the controls to adapt as the environment changes around the user. Specifically, the manipulations provide for

- the remapping of a portion of an existing user interface to a new room control, e.g., due to movement of the terminal,
- viewing of arbitrary subsets and combinations of the functionality available, and
- mixing dynamically-generated user interfaces with custom user interfaces to address inconsistencies due to platform heterogeneity.

This functionality is easily represented as operations on documents containing the associations between between programs/UIs from the services they reference, exactly the model described above.

3 The ISL Interface Specification Language

3.1 XML

We have chosen to build atop the extensible markup language (XML) for our schema design, leveraging its allowances for the creation of custom, application-specific markup languages.

XML is an SGML subset providing self-describing custom markup in the form of hierarchical named-values and advanced facilities for referencing other documents (ala the HTML <href> tag). It is one protocol among a group that is touted as the successors to HTML. (The companion protocols are XSL for style sheets and XLL for linking mechanisms.) XML includes the ability to specify, discover, and combine a group of associated document schemata — otherwise known as document type definitions (DTDs). Examples include a growing set of metadata markup proposals such as Resource Description Format (RDF) and the Dublin Core.

Unlike HTML, the set of tags in XML in flexible; the tag syntax is defined by a document's associated DTDs. A key property of XML, then, is that it is dependent on these schema to be useful, and dependent on agreements in schema to allow interoperability. Thus, the problem of defining schema syntax (the tag set and their relationship) and agreeing on how a schema's associated "browsers" (borrowing the HTML term) semantically interpret these tags is of critical importance to XML's success.

We believe there is a natural synergy between XML's need for schemata and the specification requirements of Internet distributed object systems — the former provides a self-describing and extensible syntax with a rapidly expanding set of metadata tags; the latter provides a programming model for "Internet objects" described in XML.

3.2 ISL Usage

The key challenge in implementing our approach is defining a single schema that:

- denotes services' interfaces,
- associates relevant programs and UIs to collections of services, or, vice-versa, lists the service interfaces expected by particular programs
- can compose and decompose based on constituent elements, and
- allows for incorporation of service-specific metadata (i.e., data that should not affect existing functionality that does not expect it).

We specifiy that a single document format is shared among all entities in the system. A reference to a service looks identical to the description of the service, which allows the use of structural type matching to resolve such references. Encoding the descriptions as XML documents allows middleware entities to detect documents they may want to modify via the use of XML queries [4, 22]; thus providing substructural matching (matching against just a portion of the description) in addition

to structure matching against the entire document tree. Additionally, the use of XML allows documents to be programmatically modified using the Document Object Model (DOM) [1].

As adjuncts to servers, documents act as static interface definitions, and are analogous to CORBA object IDL descriptions or the result of introspection on a Java class. As adjuncts to clients, documents act as a stable but manipulable (composable/decomposable) format for specifying service collections and references, defining interactions between services in a collection, defining the service interfaces expected by programs and user interfaces, and storing arbitrary metadata about referents. Alongside proxies — entities that act as both a server and a client — a single document fulfills both duties.

3.3 ISL Syntax

Our document markup language is called ISL, an acronym for "Interface Specification Language." We use six tags in our initial minimal design. Other tags that appear in our documents are assumed to be application-specific metadata, and can be ignored by programs that do not understand them. We now describe each tag in turn. The ISL DTD is provided in the Appendix A.

The <service> tag is a container tag. It has one optional attribute, "name", which is either a string or reference identifying the type/class of the interface being described. It can contain at most a single <label> tag, zero or one <addrspec> tags, any number of <ui> tags, and any number of <method> tags. When converted to a user interface, an <service> is instantiated as a container for widgets (a "frame").

The <label> tag provides a text description of the service which contains it. It has no optional attributes. It can contain no additional internal tags except those providing text formatting. When converted to a user interface, the <label> tag is used as a title for its parent service's frame.

The <addrspec> (address specification) tag indicates the address and port number on which its parent service listens for method invocations and events. Instantiating a service causes this tag to be added to its description; a service that has not been allocated (and thus has no addrspec) is called "unpinned." The tag can contain no additional internal tags and does not have any optional attributes. When converted to a user interface, the <addrspec> tag is used as the location to which any method calls are sent (currently via string-based UDP messages to facilitate ease of multiplexing, multicast support, and a degree of language/system independence).

The <method> tag defines the name of a method that can be invoked on the service in which it is contained. It has two optional attributes: "name", which is name of the method call, and "lexType", which indicated the lexical type of messages returned due to the method call (the list of lexical types is described below). The <method> tag can include (only) zero or more <param> tags. When used in automatic user interface generation, each <method> tag is mapped to a frame with contents. The name of the method is placed on a button at the top of this frame; pressing this button invokes the method call. Method invocations and returns are asynchronous, event-based messages rather than blocking remote procedure calls. Thus, update events ("replies") can actually occur at any time, independent of the manual invocations at the client. In this manner, <method> tags can also be used as a means for subscribing to pushed updates from a service.

The <param> tag indicates a parameter to the <method> tag that encloses it. It has two optional attributes, "lexType", indicating the lexical type of the parameter, and "optional", a boolean tag that indicates whether the parameter is required or optional. The <param> tag may have no additional internal tags, and its contents are assumed to be the name of the parameter. For UI generation, parameters are mapped to individual user interface widget objects. Widget objects are used for user input and to marshall the parameters for method invocations. Mapping from lexical type to UI widgets is described in Section 4.

The <ui> tag is used to associate a particular program to the service in which it is specified. It is unique in that there is nothing analogus to it in conventional server-side IDLs — it is useful only for clients and proxies. The contents of the tag string indicates either the name of an existing user interface object (assumed to be known or discoverable out-of-band) that will reference the document, or the address and port number from where such a user interface object can be downloaded. It has one possible attribute, "lang," indicating the language of the indicated program. There can be multiple UI tags for each service, at all levels of the service description hierarchy in an ISL document. To work with our framework, the indicated applications need to reference the document (e.g., add their own interface descriptions to it if they choose to expose one), thereby respecting the indirection exposed by the document-based approach.

4 User Interface Generation

4.1 Basic Approach

Many of the mechanics of generating user interfaces from interface descriptions were described in the preceeding section. The remaining features to be discussed are the heuristic mapping from lexical types to user interface widgets, and how custom user interfaces indicated by a <ui> tag can be intermingled with these custom user interfaces.

Dynamic user interface generation is only useful in a limited number of application domains: there is limited internal state maintenance and a lack of protocol transitions. Though both of these limitations can be addresses through additional markup, doing so blurs the distinction between the declarative nature of current design and traditional full-featured scripting languages.

We currently have implemented mappings only from primitive lexical types to objects wrapped around Tk [21] UI widgets in the MASH toolkit [15]. Permissable lexical types include int, real, boolean, enum, string. The int and real type can have an optional range modifier. They are mapped to widgets as follows: an int or real with a "range" modifier is mapped to a scale widget (a slider). Without a range modifier, they are mapped to an entry widget (a type-in box). A boolean is mapped to a checkbutton widget (a toggle switch). An enum is mapped to a list of radio-buttons (one-of-N list selection). A string is mapped to an entry widget. Structured types are expected to be expressed as (possibly hierarchical) collections in/of <service> tags, treated as aggregates through the structural type matching.

Co-mingling generated collections and existing UIs referenced in <ui> tags is done at a granularity of individual services. Thus, all services receive a frame, and it is filled with either the custom-generated contents mapped from <method> and <param> tags, or the existing UI. The latter is handed a handle to this window and is expected to instantiate itself as a child of that window.

4.2 Return Values

One approach to dealing with return values is to require per-method output type descriptions, and require that components respect these output descriptions. In this case, separate entities may be required on the reverse path to remap this data is there are type mismatches. We have not incorporated this extension into our software but believe it to be a straightforward extension. Instead, we simplify things by forcing the input and output types to be identical — a form of call-by-reference for all the arguments. This avoids the need for output specifications entirely. The simplified approach has the important benefit that reverse (response) paths can be the same as the forward (invocation) paths, just in opposite order. In the more general case, a completely separate response path might be needed, which both complicates the problem of attempting to set up such a path and adds more elements that must be checked by the user for semantics preservation. An addition important advantage is that it allows all the widgets on automatically generated user interfaces to be updated directly from the contents of the method call results, a convenient mechanism given our focus on control applications (which are amenable to use with automatic UI generation because).

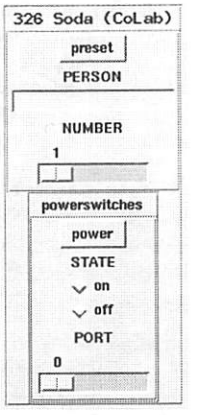
5 The Framework in Action

We now illustrate some examples. Each highlights a different element of the design of the overall architecture. We limit the scope of the examples to a single (varying) collection of services being referenced by a single (varying) user interface. Conceptually, though, the framework is amenable for use with transformational/proxy entities that are both referenced as a service by a user interface and perform references to other services to fulfill the incoming request.

The first example shows an XML document that describes the interface to a portion of the functionality available in Soda Hall's "CoLab" ("Collaboration Laboratory," borrowing Xerox PARC's terminology) and the resulting automatically generated user interface to it, as shown in Figure 2. The document describes two services, one contained in the other. The outer service implements a method for setting a preset for the entire room; the inner service is one of the services referenced by the outer one (i.e., one of the things affected by the preset) — an interface to a pair of power switches in the room. These two services, though notated and used in this hierarchical manner in this example, can also be controlled independently of one another. The <param> tags contain various lexical types, illustrating our use of heuristic mapping to widgets. This utility of this functionality is that it allows users the possibility to interact with dynamically discovered services where otherwise there is no available client program.

```
<service name="326">
        <label> Soda (CoLab) </label>
        <addrspec>spade.cs.berkeley.edu/0001</addrspec>
        <method name='preset'>
                <param lextype="string"> person </param>
                <param lextype="int:range=1-8"> number </param>
        </method>
        <service name='powerswitches'>
                <label>powerswitches</label>
                <addrspec>spade.cs.berkeley.edu/0002</addrspec>
                <method name='power'>
                        <param lextype="enum:on,off"> state </param>
                        <param lextype="int:range=0-1"> port </param>
                </method>
        </service>
</service>
```

(a) XML document



(b) User interface

Figure 2: An example document and generated user interface.

The second example illustrates combining a downloaded user interface with a generated one. The document is identical to that in the previous example except a single new tag is added: a <ui> tag to the internal (power switch) service, as shown in Figure 3(a). This causes that service's interface to be replaced by the UI object referenced in the tag rather than generated on-the-fly. The resulting difference is illustrated in Figure 3(b). This example illustrates how dynamic extensions to existing applications can be seamlessly incorporated using our architecture, a form of "plug-in" architecture similar to that used in Photoshop.

The third example illustrates use of the indirection exposed by our "document-based" model. A referent under a multiple-service <ui> tag is replaced. The document fragment shown in Figure 4(a) is assumed to be used by an existing application. The user interface for this application is a custom-designed monolithic interface referenced in the topmost <ui> tag. In Figure 4(b), one of the component services in the container has been replaced. Because the type of the referenced service remains the same, only the <addrspec> tag changes. The result of this change is that the application looks the same, but a portion of it now references a new service. This function illustrates the possibility for remapping interfaces due to, e.g., terminal mobility or fault tolerance. Specifically, the example takes a portion of the document describing the interface to the 405 Soda Hall seminar room and remaps the light switch to the one in the CoLab.

The fourth and final example illustrates the ability to easily specify the use a subset of the available functionality. The document in Figure 5(a) is the same as that from Figure 4, except all the internal services referenced from the outermost container object have been omitted. The resulting user interface is presented in Figure 5(b). This example shows how a user can easily elide material not considered relevant or not frequently used. In this case, we leave only the interface to the light switch exposed, simulating the case where the user has chosen to save screen real estate because, e.g., controlling only the lights is the most common usage.

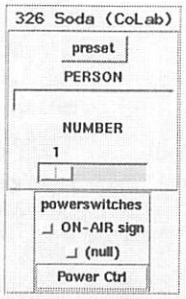
6 The User Environment

In addition to building software to parse ISL and generate appropriate interfaces, we need to provide the user with a way to manage the set of documents and available interfaces. We provide this functionality through use an "index" application, called "UC" (for "universal client"), and shown in Figure 6. One the left side of the application, all services are listed by "type" and address specification. Each type has an associated document and an associated user interface. When one of the checkbuttons beside an service name is set, the associated user interface is displayed for use by the user. Locally edited files are listed in the index with their name preceded by a hyphen. This figure illustrates a case where the user has selected to interact with three services through two user interfaces. The SodaLights UI is composed of a native light application for the Soda Hall CoLab and an automatically-generated UI for room 405. The CameraUI is another native MASH shell user interface.

The Index application provides a front-end a service discovery service (e.g., [2, 27]). Once a component is discovered, if a client-side user interface exactly matching the component name is available, it is used when then

```
<service name="326">
        <label> Soda (CoLab) </label>
        <addrspec>spade.cs.berkeley.edu/0001</addrspec>
        <method name='preset'>
                <param lextype="string"> person </param>
                <param lextype="int:range=1-8"> number </param>
        </method>
        <service name='powerswitches'>
                <label>powerswitches</label>
                <addrspec>spade.cs.berkeley.edu/0002</addrspec>
                <ui lang=mash> PowerSwitchUI </addrspec>
                <method name='power'>
                        <param lextype="enum:on,off"> state </param>
                        <param lextype="int:range=0-1"> port </param>
                </method>
        </service>
</service>
```

(a) XML document



(b) User interface

Figure 3: An example document and associated user interface, this time where a <ui> tag allows for the incorporation of a custom UI in addition to the generated components.

component is selected. If not, then the components' ISL file is downloaded (component advertisements must include a URL that points to its associated ISL file) and a user interface is generated automatically when selected. In our current prototype, other operations must be affected via manual editing of the documents; we are in the process of creating helper applications that automate common manipulations such as our cannonical "remap to local room controls" example.

7 Related Work

This approach is quite similar in flavor to the use of shell scripting to associate arbitrary programs using the UNIX pipe facility [14]. Instead of using just file handles (i.e., stdin, stdout, stderr) to differentiate and route data, structured data and cross-network references can be expressed in a document.

The Configurable Chrome group [17] of the Mozilla.org open source project is investigating a related effort. This project aims to allow users to add buttons with arbitrary functions to the Mozilla web browser (outside the main browser window) via specifications notated in a XUL document [16]. Similarly, the WinAmp MP3 client provides facilities for customizing the user interface via changing widget bitmaps, or "skins" [29]. Our approach is similar to these two approaches in the use of a document as a UI description, but adds the ability to reference full client-side interfaces in the documents rather than just widgets, and proposes that such descriptions be manipulated as the program runs.

The TSpaces project at IBM Almaden had proposed MoDAL [5] as an XML-based interpreted application description language for mobile Internet devices. In the system, documents includes such information as screen size, button tags, and label tags, and there is a two-level hierarchy: MoDAL applications are downloaded to devices running the MoDAL interpreter. In contrast, we espouse using documents as peers to application programs; i.e., there is 1) a "platform" which contains the interpreter and class libraries, 2) a traditional language (i.e. a scripting language) layer amenable to wiring together platform components, creating custom user interfaces, and being passed around on the network [15], and 3) application documents that describe interfaces and how opaque programs are combined.

8 Continuing Work

A logical next step of this work is dealing with mismatched service "types." For example, assume a light switch in some locale implements a different interface than the one in the user's home environment. Rather than require the use of a dynamically-generated user interface, we'd prefer to allow for the use of an existing user-interface. To do so, we must transparently remap method invocations to the new location and also remap the call parameters to match the new type. Incorporating such functionality allows far more flexibility in the reuse of existing user interfaces and intermingling of

```
<service name="405">
        <label> 405 Soda (HTSR) </label>
        <addrspec>htsr.cs.berkeley.edu/0000</addrspec>
        <ui lang='tcl/tk'>htsr.cs.berkeley.edu/6903</ui>
        <service name='lights'>
                <label>lights</label>
                <addrspec>htsr.cs.berkeley.edu/6902</addrspec>
                <method name='power'>
                        <param lextype="enum:on,off,dim"> state </param>
                </method>
        </service>
        ...
</service>
```

(a) Original XML document

```
<service name="405">
        <label> 405 Soda (HTSR) </label>
        <addrspec>htsr.cs.berkeley.edu/0000</addrspec>
        <ui lang='tcl/tk'>htsr.cs.berkeley.edu/6903</ui>
        <service name='lights'>
                <label>lights</label>
                <addrspec> spade.cs.berkeley.edu/9999 </addrspec>
                <method name='power'>
                        <param lextype="enum:on,off,dim"> state </param>
                </method>
        </service>
        ...
</service>
```

(b) Document with replaced referent

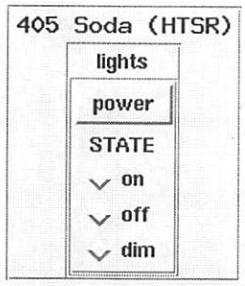
Figure 4: Remapping of function by replacing a referent under a multiple-service <ui> tag. A fragment of the "original" document is show in (a); the modified document is shown in (b), where the only difference is the new <addrspec> tag. (The <addrspec> tags are highlighted.)

existing interfaces and discovered services, but requires the use of external transformational operators that provide type coercion for method calls. Fortunately, such transformational operators could be written once, reused, and shared among the community of users; additionally, they could be chained together in order to provide new type-to-type coercions [6]. This functionality is a natural extension of our framework. We are in the process of implementing it by applying the document type conversion approach of Ockerbloom [20] to interface conversion in a system comprised of heterogeneous distributed components. The approach allows the underlying system to evolve without forcing agreement on particular component interfaces. I.e, in the language of documents, without requiring a single specific intermediate format per document style. Instead, independent pairwise conversions can be combined in chains ("paths" using the language of Ninja [18]) to allow end-to-end interoperability amongst semantically matching – but differently typed – components. Matching transformational operators are determined through the use of evolutionary structural type matching of components [25], and implement a form of wrappering [23] to encapsulate one interface in a form usable by another. Additionally, because such transformational operators can only match based on structure, there is the potential for semantic discrepancies. To address this, our approach is to require users to "bless" the use of a particular mapping operator(s) for use in particular situations. Though such a function cannot be automated (i.e, a computer cannot understand the semantics described in a textual documentation), the overhead can be reduced by allowing such decisions to be made once and shared among groups of users.

The difficulty of this approach is not in creating these mapping operators and storing them in a shared repository, but instead that of building the use of them into the end-user software. Users should be able to visually manipulate service mappings and the correct transformations should be done automatically. As a concrete example, this means that when a new light switch is discovered, the user should be able to indicate which program element should manipulate it, and any required remapping of method calls — i.e., document manipulations — should be done automatically, though possibly heuristically. Additionally, users should then be able to

```
<service name="405">
        <label> 405 Soda (HTSR) </label>
        <addrspec>htsr.cs.berkeley.edu/0000</addrspec>
        <service name='lights'>
                <label>lights</label>
                <addrspec>htsr.cs.berkeley.edu/0000</addrspec>
                <method name='power'>
                        <param lextype="enum:on,off,dim"> state </param>
                </method>
        </service>
</service>
```

(a) XML document



(b) User interface

Figure 5: Subsetting functionality. The example illustrates how functionality can be aggregated or subsetted by modifying the document associated with a program. The full description of the interface to 405 Soda has been cut down so that only a single service remains. The user interface is updated accordingly.

easily modify these mappings.

Another important extension of this work is designing how to notate one service's use of other so as to allow for the "proxy" transformational services described above; this requires separation of input and output interface descriptions. This requires extension and modification of our schema to allow such notation and extension to the software to utilize it.

Yet another area under investigation is the need for output type descriptions as described in Section 4. Similarly, to support the evolvable structural type matching of [25], we will need to add "ignorable" and "optional" attributes.

Finally, in order to allow for programmers to more easily use this document-based model — without having to manually create interface description documents — we would like to automatically generate the documents from existing Java objects and other distributed component system pieces. To do so, we can leverage the CORBA Interface Definition Language (IDL) and Java reflection API to create descriptions in our XML schema. ISL will need to be extended to support structured parameter types in order to allow such a mapping, a straightforward, but clearly important, extension.

9 Summary

Large-scale federation (composition, management, and control) of distributed Internet components requires reconsideration of how applications are structured; an ideal is to allow manipulation in response to changes in needs or component availability while not burdening application designers with details they may not require. We argue that distributed object models used with traditional application development techniques are insufficient for this purpose because they make remote object references look like static, unchanging pointers. Instead, changes should be affected at the level of a middleware layer (as they are now), but also allowed to be communicated upward to the applications in a manner allowing the applications to either ignore such changes or use them. This paper proposes a new model, a *document-based* framework, for description and interaction with Internet services. We describe a simple version of the framework in the context of remote control applications, illustrating how it allows for:

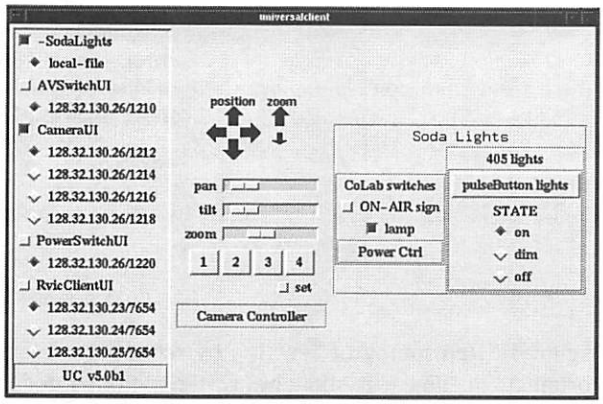


Figure 6: The Index application lists all locally available interfaces and allows the user to interactively select which ones he or she wishes to use. Illustrated here, the user has selected the aggregate user interface to some power switches and an interface to a remote-controllable camera.

- the remapping of a portion of an existing user interface,

- viewing of arbitrary subsets and combinations of functionality, and
- mixing dynamically-generated user interfaces with existing user interfaces.

The use of a document-based framework exposes an indirection between programs/UIs and the remote objects to which they refer, making this mapping explicitly manipulable, and can be used to generate user interfaces when custom ones are not available or unacceptable. It also forms the basis for continuing work on addressing interface heterogeneity through the use of structural typing, paired with wrappering in "proxy" transformational operators.

To implement our scheme, we use an XML-based language, ISL, and accompanying software. ISL:

- denotes services' available functionality, or interface, in a manner designed for interpretation and ease of manipulation at clients,
- can flexibly compose and decompose based on constituent elements, and
- allows for easy incorporation of service-specific meta-data via the self-describing, extensible nature of XML.

10 Availability

The software described herein is available in prototype form as part of the MASH toolkit, which can be downloaded from `http://www-mash.cs.berkeley.edu/mash/`.

11 Acknowledgments

The authors would like to thank the students, faculty, and staff of the MASH, Ninja, and Iceberg projects at UCB. Thanks to Michelle Munson at IBM Almaden for fruitful late-night discussions on future directions. We would also like to thank the anonymous reviewers, whose detailed commentary led to improvements in this paper (we hope). This work was supported in part by grants from Ericsson, Intel, Sprint, and Motorola, DARPA through contract DABT63-98-C-0038, the NSF through infrastructure grant CDA 94-01156, and the California MICRO program.

A Schema DTD

The document type definition (DTD) for the inital, minimal version of ISL is as follows:

```
<!ELEMENT service (label?, addrspec?, ui*,
                   method*, service*)>
<!ATTLIST service
   name  CDATA   #REQUIRED>
<!ELEMENT method (param*)>
<!ATTLIST method
   name  CDATA   #REQUIRED>
<!ELEMENT param (#PCDATA)>
<!ATTLIST param
   name  CDATA   #REQUIRED
   lexType  (int | real | boolean | enum
                | string | ...) 'string'
   optional  #BOOLEAN>
<!ELEMENT label (#PCDATA)>
<!ELEMENT addrspec (#PCDATA)>
<!ELEMENT ui (#PCDATA)>
```

References

[1] Tim Bray and Lauren Wood. The W3C Document Object Model (DOM) – A Programmer's View of Documents. *The Gilbane Report on Open Information and Document Systems*, 6(4):1–13, 1998.

[2] Steven Czerwinski, Ben Zhao, Todd Hodes, Anthony Joseph, and Randy Katz. An Architecture for a Secure Service Dis covery Service. In *Proceedings of the Fifth Annual International Conference on Mobile Computing and Networking*, Seattle, WA, August 1999. ACM.

[3] Michael L. Dertouzos. The Future of Computing. *Scientific American*, August 1999.

[4] Alin Deutsch, Mary Fernandez, Daniela Florescu, Alon Levy, and Dan Suciu. XML-QL: A Query Language for XML, August 1998. `http://www.w3.org/TR/1998/NOTE-xml-ql-19980819/`.

[5] K. Eustice, T. Lehman, A. Morales, M. C. Muson, S. Edlund, and M. Guillen. A Universal Information Appliance. *IBM Systems Journal*, pages 454–474, August 1999. (to appear).

[6] A. Fox, S. Gribble, Y. Chawathe, and E. Brewer. Adapting to Network and Client Variation Using Active Proxies: Lessons and Perspectives,. *IEEE Personal Communications, special issue on adaptation*, August 1998.

[7] Armando Fox. *A Framework for Separating Server Scalability and Availability from Internet Application Functionality*. PhD thesis, University of California, Berkeley, 1998.

[8] Hewlett-Packard. e-Speak White Paper. `http://www.hp.com/go/e-speak/`, 1997.

[9] Todd Hodes, Randy Katz, E. Servan-Schreiber, and Larry Rowe. Composable Ad hoc Mobile Services for Universal Interaction. *Proceedings of the 3rd ACM International Conference on Mobile Computing and Networking*, pages 1–12, September 1997.

[10] Todd Hodes, Mark Newman, Steve McCanne, James Landay, and Randy Katz. Shared Remote Control of a Video Conferencing Application. *SPIE Multimedia Computing and Networking*, pages 17–28, January 1999.

[11] InfoSpheres. The InfoSpheres Project. `http://infospheres.cs.caltech.edu`.

[12] G. Krasner and S. T. Pope. A Cookbook for Using the Model View Controller User Interface Paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*, August/September 1988.

[13] David Krieger and Richard Adler. The Emergence of Distributed Component Platforms. *IEEE Computer Magazine*, pages 43–53, March 1998.

[14] S. J. Leffler, M. K. McKusick, M. J. Karels, and J. S. Quarterman. *The Design and Implementation of the 4.3 BSD UNIX Operating System*. Addison-Wesley, Reading, MA, Nov 1989.

[15] Steven McCanne et al. Toward a Common Infrastructure for Multimedia-Networking Middleware. *Proc. 7th Intl. Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV '97)*, May 1997.

[16] Mozilla.org. Introduction to a XUL (XML-based User Interface Language) Document. `http://www.mozilla.org/xpfe/xptoolkit/xulintro.html`.

[17] Mozilla.org. Mozilla Configurable Chrome. `http://www.mozilla.org/aurora/config.htm`.

[18] Ninja. The Ninja Project. `http://ninja.cs.berkeley.edu`.

[19] Object Management Group. Common Object Request Broker Architecture. `http://www.omg.org/`.

[20] John Mark Ockerbloom. *Mediating Among Diverse Data Formats*. PhD thesis, Carnegie-Mellon University, 1998.

[21] J. K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley Publishing Company, Reading, MA, 1994.

[22] Jonathan Robie, Joe Lapp, and David Schach. XML Query Language (XQL). In *QL '98 - The Query Languages Workshop*, December 1998.

[23] M. T. Roth and P. Schwarz. Don't Srap It, Wrap It! A Wrapper Architecture for Legacy Data Sources. *Proceedings of 23rd VLDB Conference*, 1997.

[24] Bill N. Schilit, Norman I. Adams, and Roy Want. Context-Aware Computing Applications. In *Proceedings of the Workshop on Mobile Computing Systems and Applications*, pages 85–90. IEEE Computer Society, December 1994.

[25] Mike Spreitzer and Andrew Begel. More Flexible Data Types. *IEEE Eighth International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprisese*, 1999.

[26] Sun Microsystems. Enterprise Java Beans. `http://java.sun.com/ejb`.

[27] J. Veizades, E. Guttman, C. Perkins, and S. Kaplan. *Service Location Protocol Internet Draft #17, draft-ietf-svrloc-protocol-17.txt*. IETF, 1997.

[28] Jim Waldo. The Jini Architecture for Network-centric Computing. *Communications of the ACM*, pages 76–82, July 1999.

[29] WinAmp. WinAmp skins. `http://www.winamp.com/skins/`.

[30] World Wide Web Consortium. eXtensible Markup Language. http://w3c.org/XML/.

[31] P. Wyckoff, S. McLaughry, T. Lehman, and D. Ford. T Spaces. *IBM Systems Journal*, 37(3):454–474, August 1998.

Sting: a TCP-based Network Measurement Tool

Stefan Savage
Department of Computer Science and Engineering
University of Washington, Seattle
savage@cs.washington.edu

Abstract

Understanding wide-area network characteristics is critical for evaluating the performance of Internet applications. Unfortunately, measuring the end-to-end network behavior between two hosts can be problematic. Traditional ICMP-based tools, such as `ping`, are easy to use and work universally, but produce results that are limited and inaccurate. Measurement infrastructures, such as NIMI, can produce highly detailed and accurate results, but require specialized software to be deployed at both the sender and the receiver. In this paper we explore using the TCP protocol to provide more accurate network measurements than traditional tools, while still preserving their near-universal applicability. Our first prototype, a tool called *sting*, is able to accurately measure the packet loss rate on both the forward and reverse paths between a pair of hosts. We describe the techniques used to accomplish this, how they were validated, and present our preliminary experience measuring the packet loss rates to and from a variety of Web servers.

1 Introduction

Measuring the behavior between Internet hosts is critical for diagnosing current performance problems as well as for designing future distributed services. Unfortunately, the Internet architecture was not designed with performance measurement as a primary goal and therefore has few "built-in" services that support this need [Cla88]. Consequently, measurement tools must either "make do" with the services provided by the Internet, or deploy substantial new infrastructures geared towards measurement.

In this paper, we argue that the behavior of the commonly deployed Transmission Control Protocol (TCP) can be used as an implicit measurement service. We present a new tool, called *sting*, that uses TCP to measure the packet loss rates between a source host and some target host. Unlike traditional loss measurement tools, sting is able to precisely distinguish which losses occur in the forward direction on the path to the target and which occur in the reverse direction from the target back to the source. Moreover, the only requirement of the target host is that it run some TCP-based service, such as a Web server.

The remainder of this paper is organized as follows: In section 2 we review the current state of practice for measuring packet loss. Section 3 contains a description of the basic loss deduction algorithms used by sting, followed by extensions for variable packet size and interarrival times in section 4. We briefly discuss our implementation in section 5 and present some preliminary experiences using the tool in section 6.

2 Measuring packet loss

The rate at which packets are lost can have a dramatic impact on application performance. For example, it has been shown that for moderate loss rates (less than 15 percent) the bandwidth delivered by TCP is proportional to $1/\sqrt{lossrate}$ [MSM97]. Similarly, some streaming media applications only perform adequately under low loss conditions [CB97]. Not surprisingly, there has always been a long-standing operational need to measure packet loss; the popular `ping` tool was developed less than a year after the creation of the Internet. In the remainder of this section we'll discuss two dominant methods for measuring packet loss: tools based on the Internet Control Message Protocol (ICMP) [Pos81] and new measurement infrastructures.

2.1 ICMP-based tools

Common ICMP-based tools, such as `ping` and `traceroute`, send probe packets to a host, and measure loss by observing whether or not response packets arrive within some time period. There are two principle problems with this approach:

- *Loss asymmetry.* The packet loss rate on the forward path to a particular host is frequently quite different from the packet loss rate on the reverse path from that host. Without any additional information from the receiver, it is impossible for an ICMP-based tool to determine if its probe packet was lost or if the response was lost. Consequently, the loss rate reported by such tools is really:

  $$1 - ((1 - loss_{fwd}) \cdot (1 - loss_{rev}))$$

  Where $loss_{fwd}$ is the loss rate the forward direction and $loss_{rev}$ is the loss rate in the reverse di-

rection. Loss asymmetry is important, because for many protocols the relative importance of packets flowing in each direction is different. In TCP, for example, losses of acknowledgment packets are tolerated far better than losses of data packets. Similarly, for many streaming media protocols, packet losses in the opposite direction from the data stream have little or no impact on overall performance. The ability to measure loss asymmetry allows a network engineering to more precisely locate important network bottlenecks.

- *ICMP filtering.* ICMP-based tools rely on the near-universal deployment of the *ICMP Echo* or *ICMP Time Exceeded* services to coerce response packets from a host [Bra89]. Unfortunately, malicious use of ICMP services has led to mechanisms that restrict the efficacy of these tools. Several host operating systems (e.g. Solaris) now limit the rate of ICMP responses, thereby artificially inflating the packet loss rate reported by `ping`. For the same reasons many networks (e.g. microsoft.com) filter ICMP packets altogether. Some firewalls and load balancers respond to ICMP requests on behalf of the hosts they represent, a practice we call *ICMP spoofing*, thereby precluding real end-to-end measurements. Finally, at least one network has started to rate limit all ICMP traffic traversing it. It is increasingly clear that ICMP's future usefulness as a measurement protocol will be reduced [Rap98].

2.2 Measurement infrastructures

In contrast, wide-area measurement infrastructures, such as NIMI and Surveyor, deploy measurement software at both the sender and the receiver to correctly measure one-way network characteristics [Pax96, PMAM98, Alm97]. Such approaches are technically ideal for measuring packet loss because they can precisely observe the arrival and departure of packets in both directions. The obvious drawback is that the measurement software is not widely deployed and therefore measurements can only be taken between a restricted set of hosts. Our work does not eliminate the need for such infrastructures, but allows us to extend their measurements to include parts of the Internet that are not directly participating. For example, access links to Web servers can be highly congested, but they are not visible to current measurement infrastructures.

Finally, there is some promising work that attempts to derive per-link packet loss rates by correlating measurements of multicast traffic among many different hosts [CDH+99]. The principle benefit of this approach is that it allows the measurement of N^2 paths with $O(N)$ messages. The slow deployment of wide-area multicast routing currently limits the scope of this technique, but this situation may change in the future. However, even with universal multicast routing, multicast tools require software to be deployed at many different hosts, so, like other measurement infrastructures, there will likely still be significant portions of the commercial Internet that can not be measured with them.

Our approach is similar to ICMP-based tools in that it only requires participation from the sender. However, unlike these tools, we exploit features of the TCP protocol to deduce the direction in which a packet was lost. In the next section we describe the algorithms used to accomplish this.

3 Loss deduction algorithm

To measure the packet loss rate along a particular path, it is necessary to know how many packets were sent from the source and how many were received at the destination. From these values the one-way loss rate can be derived as:

$$1 - (packetsreceived/packetssent)$$

Unfortunately, from the standpoint of a single endpoint, we cannot observe both of these variables directly. The source host can measure how many packets it has sent to the target host, but it cannot know how many of those packets are successfully received. Similarly, the source host can observe the number of packets it has received from the target, but it cannot know how many more packets were originally sent. In the remainder of this section we will explain how TCP's error control mechanisms can be used to derive the unknown variable, and hence the loss rate, in each direction.

3.1 TCP basics

Every TCP packet contains a 32 bit sequence number and a 32 bit acknowledgment number. The sequence number identifies the bytes in each packet so they may be ordered into a reliable data stream. The acknowledgment number is used by the receiving host to indicate which bytes it has received, and indirectly, which it has not. When in-sequence data is received, the receiver sends an acknowledgment specifying the next sequence number that it expects and implicitly acknowledging all sequence numbers preceding it. Since packets may be lost, or reordered in flight, the acknowledgment number is only incremented in response to the arrival of an in-sequence packet. Consequently, out-of-order or lost packets will cause a receiver to issue duplicate acknowledgments for the packet it was expecting.

```
for i := 1 to n                          for each ack received
  send packet w/seq# i                     ackReceived++
  dataSent++
  wait for long time
```

Figure 1: Data seeding phase of basic loss deduction algorithm.

```
lastAck := 0                             for each ack received w/seq# j
while lastAck = 0                          lastAck = MAX(lastAck, j)
  send packet w/seq# n+1

while lastAck < n + 1
  dataLost++
  retransPkt := lastAck
  while lastAck = retransPkt
     send packet w/seq# retransPkt

dataReceived := dataSent - dataLost
ackSent := dataReceived
```

Figure 2: Hole filling phase of basic loss deduction algorithm.

3.2 Forward loss

Deriving the loss rate in the forward direction, from source to target, is straightforward. The source host can observe how many data packets it has sent, and then can use TCP's error control mechanisms to query the target host about which packets were received. Accordingly, we divide our algorithm into two phases:

- *Data-seeding.* During this phase, the source host sends a series of in-sequence TCP data packets to the target. Each packet sent represents a binary sample of the loss rate, although the value of each sample is not known at this point. At the end of the data-seeding phase, the measurement period is concluded and any packets lost after this point are not counted in the loss measurement.
- *Hole-filling.* The hole-filling phase is about discovering which of the packets sent in the previous phase have been lost. This phase starts by sending a TCP data packet with a sequence number *one greater* than the last packet sent in the data-seeding phase. If the target responds by acknowledging this packet, then no packets have been lost. However, if any packets have been lost there there will be a "hole" in the sequence space and the target will respond with an acknowledgment indicating exactly where the hole is. For each such acknowledgment, the source host retransmits the corresponding packet, thereby "filling the hole", and records that a packet was lost. We repeat this procedure until the last packet sent in the data-seeding phase has been acknowledged. Unlike data-seeding, hole-filling must be reliable and so the implementation must timeout and retransmit its packets when expected acknowledgments do not arrive.

3.3 Reverse Loss

Deriving the loss rate in the reverse direction, from target to source, is somewhat more problematic. While the source host can count the number of acknowledgments it receives, it is difficult to be certain how many acknowledgments were sent. The ideal condition, which we refer to as *ack parity* is that the target sends a single acknowledgment for every data packet it receives. Unfortunately, most TCP implementations use a *delayed acknowledgment* scheme that does not provide this guarantee. In these implementations, the receiver of a data packet does not respond immediately, but instead waits for an additional packet in the hopes that the cost of sending an acknowledgment can be amortized [Bra89]. If a second packet has not arrived within some small timeout (the standard limits this delay to 500ms, but 100-200ms is a common value) then the receiver will issue an acknowledgment. If a second packet does arrive before the timeout, then the receiver will issue an acknowledgment immediately. Consequently, the source host cannot reliably

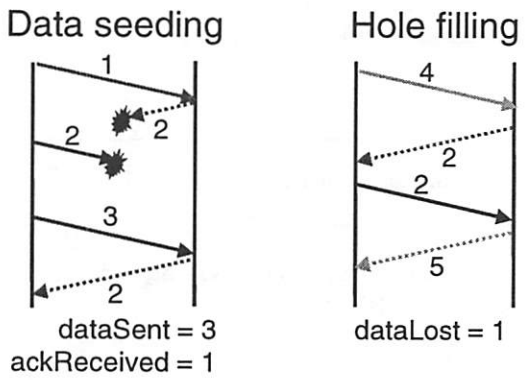


Figure 3: Example of basic loss deduction algorithm. In each time-line the left-hand side represents the source host and the right-hand side represents the target host. Right-pointing arrows are labeled with their sequence number and left-pointing arrows with their acknowledgment number.

differentiate between acknowledgments that are lost and those which are simply suppressed by this mechanism.

An obvious method for guaranteeing ack parity is to to insert a long delay after each data packet sent. This will ensure that a second data packet never arrives before the delayed ack timer forces an acknowledgment to be sent. If the delay is long enough, then this approach is quite robust. However, the same delay limits the technique to measuring packet losses over long time scales. If we wish to investigate shorter time scales, or the correlation between the sending rate and observed losses, then this mechanism is insufficient. We will discuss alternative mechanisms for enforcing ack parity in section 4.

3.4 Putting it all together

Figures 1 and 2 contain simplified pseudo-code for the algorithm as we've described it. Without loss of generality, we assume that the sequence space for the TCP connection starts at 0, each data packet contains a single byte (and therefore consumes a single sequence number), and data packets are sent according to a periodic distribution. When the algorithm completes, we can calculate the packet loss rate in each direction as follows:

$$Loss_{fwd} = 1 - (dataReceived/dataSent)$$

$$Loss_{rev} = 1 - (ackReceived/ackSent)$$

Figure 3 illustrates a simple example. Here, the first data packet is received, but its acknowledgment is lost. Subsequently, the second data packet is lost. When the third data packet is successfully received, the target responds with an acknowledgment indicating that it is still waiting to receive packet number two. At the end of the data seeding phase, we know that we've sent three data packets and received one acknowledgment.

In the hole filling phase, we send a fourth packet and receive an acknowledgment indicating that the second packet was lost. We record the loss and then retransmit the missing packet. The subsequent acknowledgment for our fourth packet indicates that the other two data packets were successfully received.

$$Loss_{fwd} = 1 - (2/3) = 33\%$$

$$Loss_{rev} = 1 - (1/2) = 50\%$$

4 Extending the algorithm

The algorithm we described is fully functional, however it has several unfortunate limitations, which we now remedy.

4.1 Fast ack parity

First, the long timeout used to guarantee ack parity restricts the tool to examining background packet loss over relatively large time scales. If we are interested in examining losses over shorter time scales, or exploring correlations between packet losses and packet bursts sent from the source, then we must eliminate the long delay requirement.

An alternative technique for forcing ack parity is to take advantage of the *fast retransmit* algorithm contained in most modern TCP implementations [Ste94]. This algorithm is based on the premise that since TCP always acknowledges the last in-sequence packet it has received, a sender can infer a packet loss by observing duplicate acknowledgments. To make this algorithm efficient, the delayed acknowledgment mechanism is *suspended* when an out-of-sequence packet arrives. This rule leads to a simple mechanism for guaranteeing ack parity: during the data seeding phase we skip the first sequence number and thereby ensure that all data packets are sent, and received, out-of-sequence. Consequently, the receiver will immediately respond with an acknowledgment for each data packet received. The hole filling phase is then modified to transmit this first sequence number instead of the next in-sequence packet.

4.2 Sending data bursts

The second limitation is that we cannot send large packets. The reason for this is that the amount of buffer space provided by the receiver is limited. Many TCP implementations default to 8KB receiver buffers. Consequently, the receiver can accommodate no more than five

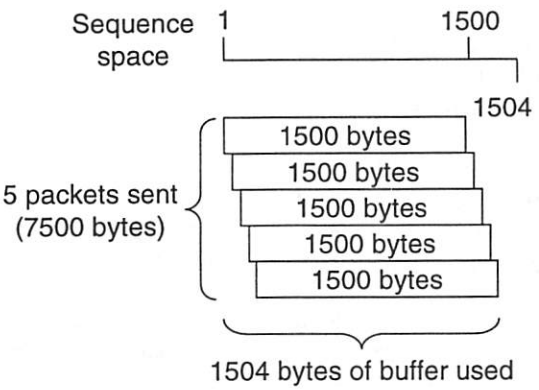


Figure 4: Mapping packets into sequence numbers by overlapping sequence numbers.

1500 byte packets, a number far too small to be statistically significant. While we could simply create a new connection and restart the tool, this limitation prevents us from exploring larger packet bursts.

Luckily, we observe that TCP implementations *trim* packets that overlap the sequence space that has already been received. Consequently, if a packet arrives that overlaps a previously received packet, then the receiver will only buffer the portion that occupies "new" sequence space. By explicitly overlapping the sequence numbers of our probe packets we can map each large packet into a single byte of sequence space, and hence only a single byte of buffer at the receiver.

Figure 4 illustrates this technique. The first 1500 byte packet is sent with sequence number 1500, and when it arrives at the target it occupies 1500 bytes of buffer space. However, the next 1500 byte packet is sent with sequence number 1501. The target will note that the first 1499 bytes of this packet have already be received, and will only use a single byte of buffer space. Using this technique we can map every additional packet into a single sequence number, eliminating much of the buffering limitation. This technique only allows us to send bursts of data in one direction – towards the target host. Coercing the target host to send arbitrarily sized bursts of data back to the source is more problematic since TCP's congestion control mechanisms normally control the rate at which the target may send data. We have investigated techniques to remotely bypass TCP's congestion control [SCWA99] but we believe they represent a security risk and aren't suited for common measurement tasks.

4.3 Delaying connection termination

One final problem is that some TCP servers do not close their connections in a graceful fashion. TCP connections are full-duplex – data flows along a connection in both directions. Under normal conditions, each "half" of the connection may only be closed by the sending side (by sending a FIN packet). Our algorithms implicitly assume this is true, since it is necessary that the target host respond with acknowledgments until the testing period is complete. While most TCP-based servers follow this termination protocol, we've found that some Web servers simply terminate the entire connection by sending a RST packet – sometimes called an *abortive release*. Once the connection has been reset, the sender discards any related state so any further probing is useless and our measurement algorithms will fail.

To ensure that our algorithms have sufficient time to execute, we've developed two ad hoc techniques for delaying premature connection termination. First, we ensure that the data sent during the data seeding phase contains a valid HTTP request. Some Web servers (and even some "smart" firewalls and load balancers) will reset the connection as soon as the HTTP parser fails. Second, we use TCP's *flow control* protocol to prevent the target from actually delivering its HTTP response back to the source. TCP receivers implement flow control by advertising the number of bytes they have available for buffering new data (called the *receiver window*). A TCP sender is forbidden from sending more data than the receiver claims it can buffer. By setting the source's receiver window to zero bytes we can keep the HTTP response "trapped" at the target host until we have completed our measurements. The target will not reset the connection until its response has been sent, so this technique allows us to inter-operate with such "ill-behaved" servers.

5 Implementation

In principle, it should be straightforward to implement the loss deduction algorithms we have described. However, in most systems it is quite difficult to do so without modifying the kernel and developing a portable application-level solution is quite a challenge. We observe that the same problem is true for any user-level implementation of TCP. The principle difficulty is that most operating systems do not provide a mechanism for redirecting packets to a user application and consequently the application is forced to coordinate its actions with the host operating system's TCP implementation. In this section we will briefly describe the implementation difficulties and explain how our current prototype functions.

5.1 Building a user-level TCP

Most operating systems provide two mechanisms for low-level network access: *raw sockets* and *packet filters*. A raw socket allows an application to directly format and send packets with few modifications by the underlying system. Using raw sockets it is possible to create our own TCP segments and send them into the network. Packet filters allow an application to acquire *copies* of raw network packets as they arrive in the system. This mechanism can be used to receive acknowledgments and other control messages from the network. Unfortunately, another copy of each packet is also relayed to the TCP stack of the host operating system; this can cause some difficulties. For example, if sting sends a TCP SYN request to the target, the target responds with a SYN of its own. When the host operating system receives this SYN it will respond with a RST because it is unaware that a TCP connection is in progress.

An alternative implementation would be to use a secondary IP address for the sting application, and implement a user-level proxy ARP service. This would be simple and straightforward, but has the disadvantage that users of sting would need to request a second IP address from their network administrator. For this reason, we have resisted this approach.

Finally, many operating systems are starting to provide proprietary firewall interfaces (e.g. Linux, FreeBSD) that allow the user to filter outgoing or incoming packets. The former ability could be used to intercept packets arriving from the target host, while the later ability could be used to suppress the responses of the host operating system. We are investigating this approach for a future version.

5.2 The Sting prototype

Our current implementation is based on raw sockets and packet filters running on FreeBSD 3.x and Digital Unix 3.2. As a work-around to the SYN/RST problem mentioned previously, we use the standard Unix `connect()` service to create the connection, and then hijack the session in progress using the packet filter and raw socket mechanisms. Unfortunately, this solution is not always sufficient as the host system can also become confused by acknowledgments for packets it has never sent. In our current implementation we have been forced to change one line in the kernel to control such unwanted interactions.[1] We are currently unsure if a completely portable user-level implementation is possible on today's Unix systems.

Figure 5 shows the output presented by sting. From the command line the user can select the inter-arrival distribution between probe packets (periodic, uniform, or exponential), the distribution mean, the number of total packets sent, as well as the target host and port. Our implementation verifies that the wire time distribution conforms to the expected distribution according to the tests provided in [PAMM98].

```
# sting www.audiofind.com

Source = 128.95.2.93
Target = 207.138.37.3:80
dataSent = 100
dataReceived = 98
acksSent = 98
acksReceived = 97
Forward drop rate = 0.020000
Reverse drop rate = 0.010204
```

Figure 5: Sample output from the sting tool. By default, sting sends 100 probe packets according to a uniform inter-arrival distribution with a mean of 100ms.

We have tested our implementation in several ways. First, we have synthetically dropped packets in the tool and using an emulated network [Riz97] and verified that sting reports the correct loss rate. Second, we have compared the results of sting to results obtained from `ping`. Using the derivation for `ping`'s loss rate presented in section 2 we have verified that the the results returned by each tool are compatible. Finally, we have tested sting with a large number of different host operating systems, including Windows 95, Windows NT, Solaris, Linux, FreeBSD, NetBSD, AIX, IRIX, Digital Unix, and MacOS. While we occasionally encounter problems with very poor TCP implementations (e.g. laser printers) and Network Address Translation boxes, the tool is generally quite stable.

6 Experiences

Anecdotally, our experience in using sting has been very positive. We've had considerable luck using it to debug network performance problems on asymmetric access technologies (e.g. cable modems). We've also used it as a day-to-day diagnostic tool to understand the source of Web latency. In the remainder of this section we present some preliminary results from a broad experiment to quantify the character of the loss seen from our site to the rest of the Internet.

For a twenty four hour period, we used sting to record loss rates from the University of Washington to a collection of 50 remote web servers. Choosing a reasonably-sized, yet representative, set of server sites is a difficult task due to the diversity of connectivity and load expe-

[1] We modify the ACK processing in tcp_input.c so the response to an acknowledgment entirely above snd_max is to drop the packet instead of acknowledging it.

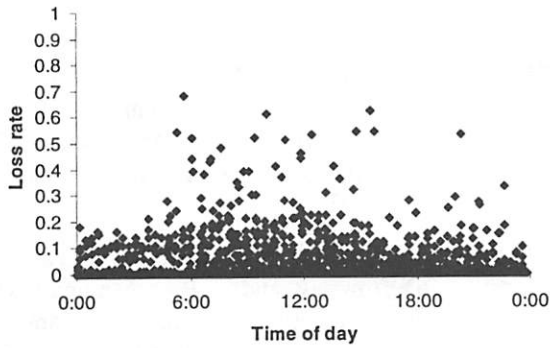


Figure 6: Forward loss measured across a twenty four hour period. Each point on this scatter-plot represents a measurement to one of 50 Web servers.

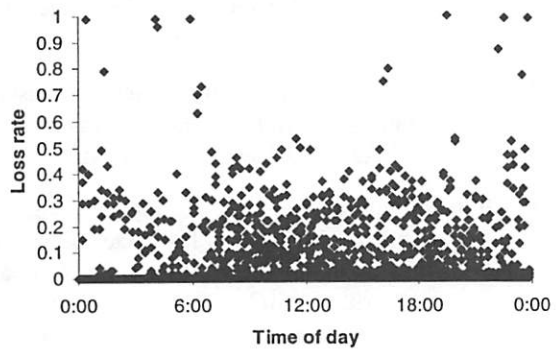


Figure 7: Reverse loss measured across a twenty four hour period. Each point on this scatter-plot represents a measurement to one of 50 Web servers.

rienced at different points in the Internet. However, it is well established that the distribution of Web accesses is heavy-tailed; a small number of popular sites constitute a large fraction of overall requests, but the remainder of requests are distributed among a very large number of sites [BCF+99]. Consequently, we have constructed our target set to mirror this structural property – popular servers and random servers. Half of the 50 servers in our set are chosen from a list of the top 100 Web sites as advertised by `www.top100.com` in May of 1999. This list is generated from a collection of proxy logs and trace files. The remaining 25 servers were selected randomly using an interface provided by Yahoo! Inc. to select pages at random from its on-line database [Yah].

For our experiment we used a single centralized data collection machine, a 200Mhz Pentium Pro running FreeBSD 3.1. We probed each server roughly once every 10 minutes.

Figures 6 and 7 are scatter-plots showing the overall distribution of loss rates, forward and reverse re-

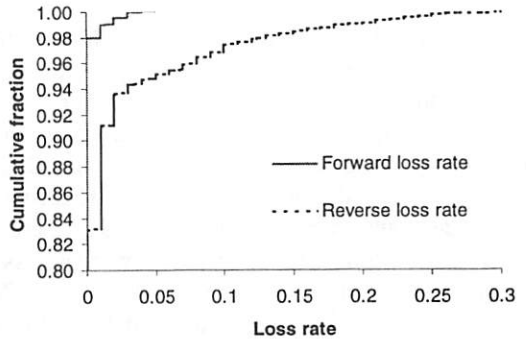


Figure 8: CDF of the loss rates measured to and from a set of 25 popular Web servers across a twenty-four hour period.

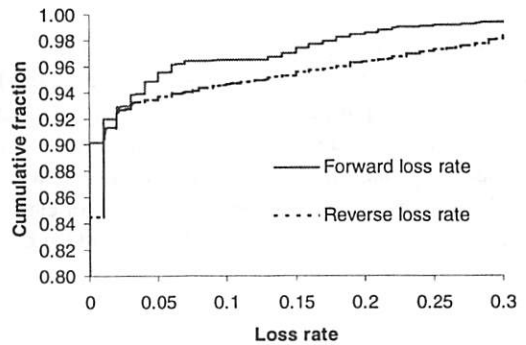


Figure 9: CDF of the loss rates measured to and from a set of 25 random Web servers across a twenty-four hour period.

spectively, during our measurement period. Not surprisingly, overall loss rates increase during business hours and wane during off-peak hours. However, it is also quite clear that forward and reverse loss rates vary independently. Overall the average reverse loss rate (1.5%) is more than twice the forward loss rate (0.7%) and at many times of the day this ratio is significantly larger.

This reverse-dominant loss asymmetry is particularly prevalent among the popular Web servers. Figure 8 graphs a discrete cumulative distribution function (CDF) of the loss rates measured to and from the 25 popular servers. Here we can see that less than 2 percent of of the measurements to these servers ever record a lost packet in the forward direction. In contrast, 5 percent of the measurements see a reverse loss rate of 5 percent or more, and almost 3 percent of measurements lose more than a tenth of these packets. On average, the reverse loss rate is more than 10 times greater than the forward loss rate in this population. One explanation for this phenomenon is that Web servers generally send much more traffic than they receive, yet bandwidth is provisioned in a full-duplex fashion. Consequently, bottlenecks are much more likely to form on paths leaving popular Web

servers and packets are much more likely to be dropped in this direction.

We see similar, although somewhat different results when we examine the random server population. Figure 8 graphs the corresponding CDF for these servers. Overall the loss rate is increased in both directions, but the forward loss rate has increased disproportionately. We suspect that this effect is strongly related to the lack of dedicated network infrastructure at these sites. Many of the random servers obtain network access from third-tier ISP's that serve large user populations. Consequently, unrelated Web traffic being delivered to other ISP customers directly competes with the packets we send to these servers.

7 Conclusion

In this paper, we have described techniques for measuring packet loss using TCP. Using these methods the sting tool can separately derive measures for packet loss along the forward and reverse paths to a host, and can be used to probe any TCP-based server. In the future we plan to develop TCP-based techniques for estimating bandwidth [LB99], and round-trip time, and for deriving queue length distributions.

Acknowledgments

We would like to thank a number of people for their contributions to this project. Neal Cardwell, Geoff Voelker, Alec Wolman, Matt Zekauskas, David Wetherall and Tom Anderson delivered valuable feedback on this paper and on the development of sting. Vern Paxson provided the impetus for this project by suggesting that it would be hard to do. USENIX has graciously provided a student research grant to fund ongoing work in this area. Finally, Tami Roberts was exceptionally generous and accommodating in supporting the author during the paper submission process.

References

[Alm97] Guy Almes. Metrics and Infrastructure for IP Performance. `http://www.advanced.org/surveyor/presentations.html`, 1997.

[BCF+99] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. Web Caching and Zipf-like Distributions: Evidence and Implications. In *Proceedings of the IEEE INFOCOM '99*, March 1999.

[Bra89] R. Braden. Requirements for Internet Hosts – Communication Layers. RFC-1122, October 1989.

[CB97] Georg Carle and Ernst W. Biersack. Survey of Error Recovery Techniques for IP-Based Audio-Visual Multicast Applications. *IEEE Network Magazine*, 11(6):24–36, November 1997.

[CDH+99] R. Caceres, N.G. Duffield, J. Horowitz, D. Towsley, and T. Bu. Multicast-Based Inference of Network-Internal Characteristics: Accuracy of Packet Loss Estimation. In *Proceedings of the IEEE INFOCOM '99*, New York, NY, March 1999.

[Cla88] D. Clark. The Design Philosophy of the DARPA Internet Protocols. In *Proceedings of the ACM SIGCOMM '88*, pages 106–114, Palo Alto, CA, September 1988.

[LB99] Kevin Lai and Mary Baker. Measuring Bandwidth. In *Proceedings of the IEEE INFOCOM '99*, New York, NY, March 1999.

[MSM97] Matthew Mathis, Jeffrey Semke, and Jamshid Mahdavi. The Macroscopic Behavior of the TCP Congestion Avoidance Algorithm. *ACM Computer Communications Review*, 27(3):67–82, July 1997.

[PAMM98] V. Paxson, G. Almes, J. Mahdavi, and M. Mathis. Framework for IP performance metrics. RFC-2230, May 1998.

[Pax96] Vern Paxson. End-to-End Routing Behavior in the Internet. In *Proceedings of the ACM SIGCOMM '96*, pages 25–38, Stanford, CA, August 1996.

[PMAM98] Vern Paxson, Jamshid Mahdavi, Andrew Adams, and Matthew Mathis. An Architecture for Large-Scale Internet Measurement. *IEEE Communications*, 36(8):48–54, August 1998.

[Pos81] J. Postel. Internet Control Message Protocol. RFC-792, September 1981.

[Rap98] Chris Rapier. ICMP and future testing (IPPM mailing list). `http://www.advanced.org/IPPM/archive/0606.html`, December 1998.

[Riz97] L. Rizzo. Dummynet: A Simple Approach to the Evaluation of Network Protocols.

ACM Computer Communications Review, 27(1), January 1997.

[SCWA99] Stefan Savage, Neal Cardwell, David Wetherall, and Tom Anderson. TCP Congestion Control with a Misbehaving Receiver. Draft, in review, 1999.

[Ste94] W. Richard Stevens. *TCP/IP Illustrated*, volume 1. Addison Wesley, 1994.

[Yah] Yahoo! Inc. Random yahoo! link url. `http://random.yahoo.com/bin/ryl`.

JPEG Compression Metric as a Quality Aware Image Transcoding

Surendar Chandra and Carla Schlatter Ellis
Department of Computer Science, Duke University, Durham, NC 27708
{surendar,carla}@cs.duke.edu

Abstract

Transcoding is becoming a preferred technique to tailor multimedia objects for delivery across variable network bandwidth and for storage and display on the destination device. This paper presents techniques to quantify the quality-versus-size tradeoff characteristics for transcoding JPEG images. We analyze the characteristics of images available in typical Web sites and explore how we can perform informed transcoding using the JPEG compression metric. We present the effects of this transcoding on the image storage size and image information quality. We also present ways of predicting the computational cost as well as potential space benefits achieved by the transcoding. These results are useful in any system that uses transcoding to reduce access latencies, increase effective storage space as well as reduce access costs.

1 Introduction

The advent of inexpensive hardware such as powerful personal computers, digital cameras, scanners and other easy-to-use technologies is making it easier for the average user to dabble in multimedia. The phenomenal growth of Internet technologies such as the Web and electronic mail allows users to disseminate and share these multimedia objects. By some estimates [19], about 77% of the data bytes accessed in the web are from multimedia objects such as images, audio and video clips. Of these, 67% of the data are transferred for images. While this ability of users to share multimedia objects makes the Internet more valuable to consumers, the underlying capabilities of the system are not always able to keep up with this shifting usage.

Users access these multimedia objects from a wide variety of devices with different resource constraints. Users are not only accessing the rich multimedia objects from traditional desktops that are better able to render these objects, but they are also using mobile devices such as a palmtops and laptops as well as newer devices such as webtops and navigation systems that are resource constrained in terms of the processing, storage and display capabilities. The network capabilities used in accessing these multimedia objects vary widely from wired networks such as high speed LANs, ISDN, DSL, cable and telephone modems, as well as wireless technologies such as cellular, CDPD, Ricochet and GSM networks. Depending on the technology used, the networks can be slow, unreliable and expensive.

In such an environment of varying network, storage and display capabilities, one size does not fit all. Consumers using expensive networks want to download multimedia images for the lowest possible cost. Consumers using high speed networks and high quality displays want to view the multimedia images at the highest quality.

In such an operating environment, transcoding can be used to serve the same multimedia object at different quality levels to the different users. Transcoding is a transformation that converts a multimedia object from one form to another, frequently trading off object fidelity for size. By their very nature, multimedia objects are amenable to soft access where the user is willing to compromise object fidelity for faster access. From the server's perspective, transcoding may be employed in an attempt to achieve greater scalability, as exemplified by AOL's across-the-board transcoding of all requested JPEG and GIF images [1].

Transcoding of multimedia objects can be performed in any of the following scenarios:

static environment where the information provider transcodes the object to a variety of formats (e.g., thumbnails along with full scale images) so that the consumer can download the appropriate form for the current operating environment.

streamed environment where the network infrastructure (e.g., a network proxy) can transcode the objects on the fly to compensate for network latencies.

Previous work by [19, 8, 11] has used transcoding in this fashion.

store and forward environment where the system can transcode the objects to a smaller size to increase the effective size of the local image storage space. Users of a digital camera may transcode a previous picture to a lower quality version to create space for a new picture.

For transcoding to be useful, we need to quantify the loss in information so that an informed decision can be made on choosing the aggressiveness of transcoding. We need to understand the computational costs and storage benefits in order to decide if a particular transcoding is worth the effort. For example, without such understanding, systems that have used JPEG Quality Factor as a transcoding metric typically transcoded the images to a drastically lower JPEG Quality Factor.

Our work addresses the problem of how to characterize the quality-size tradeoffs involved in transcoding JPEG images so that the effectiveness of a particular level of transcoding can be predicted efficiently. This implies several subproblems.

The first problem is to more precisely define what we mean by an "effective transcoding". Thus we introduce the notion of "quality aware transcoding" that is meant to capture a direct and measurable relationship between reduction in perceived image quality and savings in object size. In order to measure whether we achieve that goal for a particular image and a particular transformation, we need to be able to quantify the reduction in quality caused by the operation. This requires that we have a metric that corresponds, in some sense, to the user's perception of quality and that we can consistently measure the starting and ending quality levels in order to determine the loss. We analyze the use of the JPEG compression metric for that purpose. The size reduction component is straight-forward.

Next we must determine the computational cost required to perform the transcoding. Thus, we ask whether the transcoding is easy to compute and whether it can be streamed.

Finally, it is highly desirable to easily predict the outcome of a possible transcoding operation in terms of the size reductions, quality lost, and computation cost to determine if it is worth the effort for a particular case. Since transcoding may not provide space benefits for all images, we need to analyze if we can predict whether a particular transcoding will provide space benefits for the particular image. Thus we explore prediction algorithms that can estimate those outcomes. For our study, we analyze a number of JPEG images available from a number of Internet Web sites. This establishes how our methods apply to typical workloads (of JPEG images). Similar work has to be undertaken for other image, audio and video multimedia types.

Step	Compression	Step	Decompression
1	Convert ColorSpace	5	Convert Colorspace
2	Downsample	4	Upsample
3	Forward DCT	3	Inverse DCT
4	Quantize	2	De-Quantize
5	Entropy Encode	1	Entropy Decode

Table 1: JPEG Compression and Decompression

In this paper, we show that, for a transcoding that changes the JPEG compression metric (referred to as the JPEG Quality Factor), the change in JPEG Quality Factor directly corresponds to the information quality lost. To measure this change, we describe an algorithm to compute the Independent JPEG Group's (IJG) [15] equivalent of the JPEG Quality Factor of any JPEG image. To understand the overhead involved in performing a transcoding, we develop a predictor that predicts the computational overhead with a high degree of accuracy. We also develop a predictor for predicting if an image will transcode efficiently. We define transcoding efficiency of an transcoding algorithm by the ability to lose more in storage space for a particular loss in information quality. We show that we can predict efficient images at a significantly better rate than the base case. We validate these results with a number of JPEG images.

The results from this paper were utilized in a companion paper [4] that describes the utility of quality aware transcoding for serving multimedia objects to mobile clients.

The remainder of this paper is organized as follows: Section 2 describes the JPEG compression metric and how it can be computed for a given JFIF image. Section 3 describes the workload used for evaluating our transcoding technique. Results of transcoding the images for this workload are presented in Section 4. Section 5 discusses related work in the area of transcoding and image quality metrics and Section 6 summarizes the work.

2 JPEG Compression Metric

JPEG [20] is the Joint Photographic Experts Group lossy compression scheme for still images. JFIF [10] is the

JPEG File Interchange Format used for exchanging image files compressed using JPEG compression. JPEG compression is based on psycho-visual studies of human perception. To convert to a smaller file size, this type of compression drops the least-noticeable picture information. Human visual system response is dependent on the spatial frequency. JPEG is a lossy image compression algorithm that uses Discrete Cosine Transform (DCT). DCT provides a good approximation to allow one to decompose an image into a set of waveforms, each with a particular spatial frequency. This allows us to successively drop frequency components that are imperceptible to the human eye.

The different steps in compressing and decompressing an image using JPEG are outlined in Table 1. To compress an image, the color space of the image is first transformed to the YCbCr [2] color space, followed by any smoothing operations, followed by Minimum Code Unit (MCU) assembly and forward DCT computation, followed by quantization and entropy encoding. Since the human eye is less sensitive to chrominance values than to luminance values, different color components can be downsampled differently: Cb and Cr components are usually downsampled by a 2x1 factor, while the Y component is scaled using a 1x1 factor. Decompression process essentially reverses these steps: entropy decoding is performed, followed by dequantization and inverse DCT, followed by upscaling and color space conversion to get back to an RGB image.

In JPEG, the compression ratio of the output image is controlled solely by the quantization tables used in step 4. An all-1's quantizer achieves a compression ratio that is comparable to a lossless compression algorithm. The JPEG specification [3] section K.1, provides standard luminance and chrominance quantization tables that provide good results for a variety of images. The standard quantization tables can be scaled to vary the compression ratios. Most JPEG compressors allow the user to specify a range of values for the scaling factor, by specifying a compression metric called Quality Factor. This Quality Factor is an artifact of JPEG compression. Different software implementations use different values for Quality Factor. Quality Factors are not standardized across JPEG implementations. The IJG Library [15] uses a 0-100 scale, while Apple formerly used a scale running from 0 to 4. Recent Apple software uses an 0-100 scale that is different from the IJG scale. Paint Shop Pro's scale uses a 100-0 scale, where lower numbers imply higher quality. Adobe Photoshop gives discrete *maximum/ high/ medium/ low* choices.

The JPEG Quality Factor can form the basis for a transcoding that progressively reduces the Quality Factor of an image to achieve better compression ratios. A transcoding that uses JPEG Quality Factor, can transcode an image, either to an absolute Quality Factor value or to a percentage of the original quality.

2.1 Initial Quality Factor

In order to determine if the JPEG compression metric forms an effective transcoding, we need the ability to measure the JPEG Quality Factor of an image. Without the ability to measure the initial Quality Factor used to produce an image, the transcoding algorithm might transcode an image with a low initial Quality Factor to an apparently higher Quality Factor value by manipulating the quantization tables, even though such an operation does not increase the information quality of the output transcoded image. The resulting output image from such an operation is bigger than the original "lower quality" image. For example, transcoding a JPEG image of Quality Factor 20 and size 37 KB to a JPEG image of Quality Factor 50 produces an image of size 42 KB. Systems that have used JPEG compression metric as a transcoding metric, such as [8, 11], have avoided this problem of not knowing the initial JPEG Quality Factor by transcoding the images to a sufficiently low quality value, such as Q=5, so that they can transcode all images to a smaller size.

The quantization table that was used to compress an image is stored in the JFIF header, but the JPEG Quality Factor that was used to generate the quantization table is not stored along with the image and hence the original JPEG Quality Factor is lost. Different JPEG compressors use different quantization tables. Since there is no reliable way to recognize the software that produced a given JPEG image, we develop a predictor to measure the IJG equivalent of the JPEG Quality Factor of an image.

By default, the IJG implementation uses the standard quantization tables, computes the Scaling Factor(S) from the Quality Factor(Q), and then uses the Scaling Factor to scale the standard quantization tables (*stdQuantTbl*). Finally it computes the quantization tables for the image (*imgQuantTbl*) as follows

$$S = (Q \geq 50)?(\tfrac{5000}{Q}) : (200 - 2Q)$$
$$imgQuantTbl[i] = \tfrac{stdQuantTbl[i]*S+50}{100}$$

We reverse the steps used by IJG in computing the quantization tables to compute the quality value as follows:

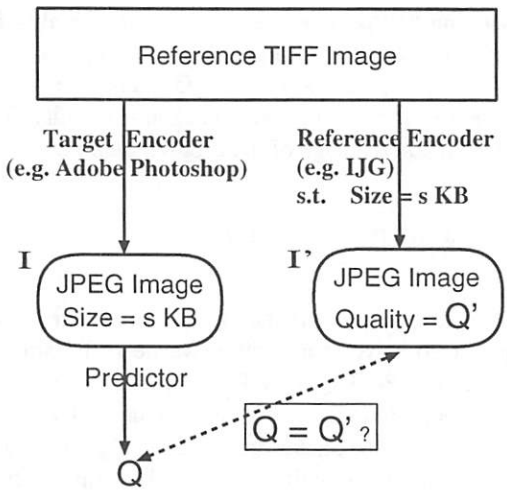


Figure 1: Evaluating JPEG Predictor

$$S' = \frac{imgQuantTbl[i] * 100 - 50}{stdQuantTbl[i]}$$
$$Q' = (S' \leq 100)?(\frac{200 - S'}{2}) : (\frac{5000}{S'})$$

The JPEG Quality Factor predictor function involves integer computations on the quantization tables that introduce integer rounding errors. Hence, even for images produced by IJG's software, the Quality Factors computed can be off by a point or two.

To validate the correctness of our JPEG Quality predictor, we first tested the predictor on images produced by IJG software. Using IJG, we created a JPEG image with a specified JPEG Quality Factor, and verified if our predictor could predict the correct Quality Factor that was used. As expected, the results were accurate within a few points; which is attributable to integer round-off errors.

Next, we tested the effectiveness in predicting the JPEG Quality of images produced by Adobe Photoshop and Paintshop Pro. We evaluated the effectiveness of our predictor in two steps. The steps are illustrated in Figure 1. First we produced a JPEG image (I) from a reference TIFF image using the target encoder (e.g. Photoshop). We ran our predictor on the resulting JPEG image to estimate its IJG-equivalent Quality Factor (Q). Then, in step two, we produced another JPEG version of the reference TIFF image (I') but used the IJG software using a Quality Factor value such that the resulting image matches the size of the JPEG image produced in Step one. Finally we compared the predicted Quality Factor and the value used in producing the IJG JPEG version. The values of Q and Q' should be identical for a successful predictor.

First we tested the effectiveness of our compressor for images produced using Adobe Photoshop. We compressed a reference TIFF image using the four quality settings allowed by Photoshop. Our predictor estimated the qualities for the four different Adobe settings to be 37, 62, 82 and 92 respectively. The sizes of the files derived from Adobe's software's four compression settings were similar to IJG compressed images of Quality Factors 55, 74, 89 and 97 respectively.

Next, we tested the effectiveness of our predictor for images produced using Paint Shop Pro. We compressed a reference TIFF image using Paint Shop for a range of Quality values (Q') between 10 through 90. Our predictor estimated the Quality Factor values at a value equal to (100 - Q'). The file sizes also corresponded to JPEG Quality values of (100 - Q'). In fact, the file size values matched so perfectly, we suspect that Paint Shop Pro internally uses IJG software, but chooses to invert the meaning of IJG Quality Factor values.

Hence, our JPEG Quality Factor predictor closely predicts the JPEG Quality Factor for images compressed using IJG and Paint Shop Pro and underestimates the JPEG Quality Factors for images compressed using Adobe Photoshop. For our purposes, it is preferable to under predict the image quality. Otherwise, applications might try to transcode an image to a Quality Factor that is lower than an over-estimated Quality Factor, but is in fact higher than the correct Quality Factor of the image (which would produce a JPEG image that is bigger than the original un-transcoded image). Since we do not know the *real* Quality Factors, and since the values are either under predicted or predicted accurately, the predictors' performance is acceptable as a predictor of the IJG equivalent of Quality Factor of the given JPEG image.

2.2 Perceived Information Quality

When we transcode an image, we want to quantify the loss in information as well as the space savings achieved. To quantify the loss in information we need to measure the image quality, as *perceived by an observer*. Then, we need to address whether JPEG Quality Factor captures perceived quality adequately.

The perceived information quality of an image depends on a variety of factors such as the viewing distance, ambient light, the display characteristics, the image size, the image background etc. Images can be objectively measured using *distortion metrics*, which are functions that find the difference between two images, *fidelity metrics*, which are functions (or series of functions) that describe the *visible* difference between two images and

subjective measures such as *quality metrics* that are numbers, derived from physical measurements, which relate to perceptions of image quality.

Distortion metrics, such as tone, color, resolution, sharpness and noise, describe a measurable change in an image, although this change need not necessarily be visible. Distortion metrics do not take into account the characteristics of either the output device or observer, although some attempts have been made to include these factors.

Fidelity metrics, such as the Visual Differences Predictor (VDP), produce difference images that map perceptible differences between two test images. While this approach is useful for diagnostic tasks, a single value representing 'quality' is often more useful. Ahumada and Null [13] note that the relationship between fidelity and quality varies with the observer's expectations of the intended use for the image, a feature which is also dependent on factors such as past experience. The subjective assessment of quality also varies with the pictorial content of the test stimuli, even when image fidelity remains constant.

In his Ph.D. thesis, Adrian Ford [7], tried a series of image quality and color reproduction measures to quantify the effects of lossy image compression on baseline JPEG images using the IJG software. Objective measurements such as tone reproduction, color reproduction (CIE $\Delta E^*$and Color Reproduction Index), resolution, sharpness (MTF) and noise (Noise Power Spectra) along with quality metrics, such as 'Barten's Square Root Integral with Noise' and 'Töpfer and Jacobson's Perceived Information Capacity' were used to evaluate the overall image quality. The image quality metrics were implemented with the aid of a published model for the human eye and a model of the display system based on experimental measurements. Measured image quality was compared with subjective quality assessments performed under controlled conditions. He concluded that JPEG compression metrics such as JPEG Quality Factor outperformed other measures in predicting the quality of an image.

Since the workload and assumptions of Adrian Ford's thesis closely matches the kinds of images available on the internet, we adopt his conclusion that JPEG Quality Factor is a good representation of the subjective image quality. Reduction in JPEG Quality Factor directly translates to loss of image information quality.

3 Workload

The effectiveness of our study depends on the realism of the JPEG images that are used to validate the results. To better understand the qualities of the JPEG images available on the web, we classify web sites into four distinct categories with respect to the site's usage of JPEG images. These categories are

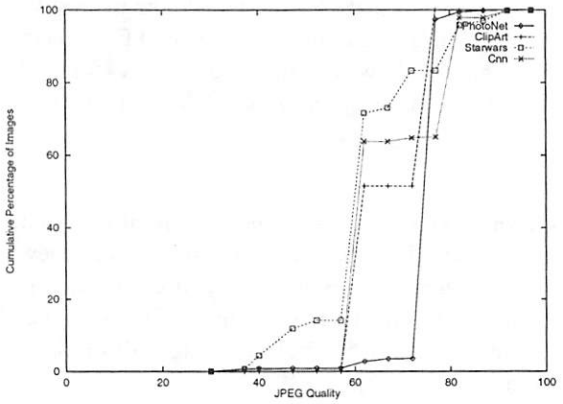
News Site. News sites use images to reinforce some news story. The images are secondary to the news being delivered. Hence we expect the images to be small and of low quality. For our experiments, we used 6217 JPEG images downloaded from Cnn.com [5].

Image Site. The sole purpose of online art gallery sites is to deliver high quality images. For our experiments, we used 4650 JPEG images downloaded from Photo.net [9].

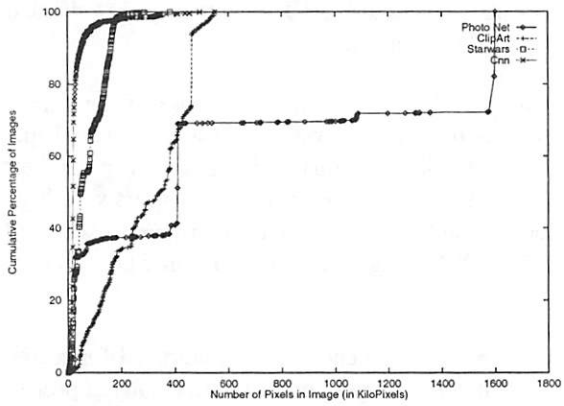
Commerce Site. The primary purpose of commerce sites is to sell their wares. These sites would like to deliver big, high quality images to promote their merchandise without turning away users with high access latencies. For our experiments, we used 1248 JPEG images downloaded from Starwars.com [22].

ClipArt Site. JPEG encoding is optimized for realistic full color images and is not especially appropriate for compressing images with few colors such as cartoons and line drawings. Still, such ill-advised uses do exist in the Web. We use 485 JPEG images from a image collection from Media Graphics [17], which provides clipart collections that are intended to be used by a Web site designer.

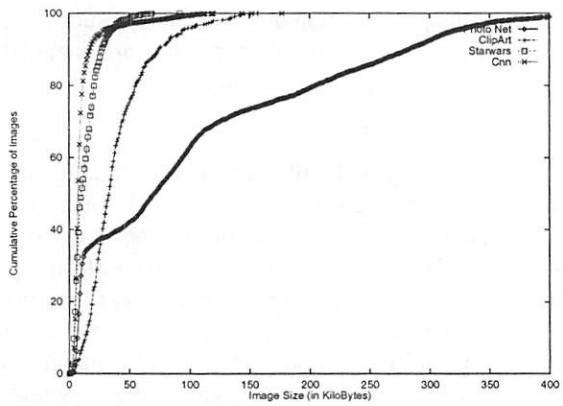
We plot the cumulative distribution of JPEG Quality, the image size and image geometry in Figure 2. From Figure 2(a), we note that 60% of images in the CNN workload have Quality Factor less than 60, while 80% of images in Starwars workload have Quality Factor less than 60. Most of the images in PhotoNet have Quality Factor of 77. From Figure 2(b), we note that most of the images in CNN and Starwars workload have small image geometries. Correspondingly, from 2(c), we note that images in the CNN and Starwars occupy smaller file sizes, with most of the images being less than 20KB. PhotoNet on the other hand has a number of very big images that are greater than 100KB. ClipArts consists of fairly big images with 25% of the images being over 50KB.



(a) Image JPEG Quality



(b) Number of Pixels in image



(c) Image file size distribution

Figure 2: Workload Characteristics

Workload	Reduced Image Quality		
	25%Q	50%Q	75%Q
Overall	3.2%	47.4%	66.1%
CNN	2.9%	40.2%	99.6%
PhotoNet	1.8%	66.3%	25.1%
ClipArt	2.3%	22.7%	53.6%
Starwars	6.8%	18.9%	56.9%

Table 2: % Efficient images for a given loss in image quality

4 Results

4.1 Efficient, Quality Aware Transcoding

Once we quantify the loss in information, we can measure if the transcoding produces significant saving in space for a corresponding loss in image information quality.

We define transcoding efficiency of a transcoding operation by the ability to lose more in size for a particular loss in information quality. For example, if an image was transcoded to lose 50% of information quality, the output image size should be less than 50% of the original image for it to be an efficient transcoding. For some applications, this restriction may be too restrictive and these applications may relax the constraint to accept output size of, say, 55% instead of 50%.

From Section 2.1, we note that our initial JPEG Quality Factor predictor under-estimates the Quality Factor of images compressed using Adobe Photoshop. As a consequence, the quality loss may be similarly underestimated which has the effect of allowing smaller size reductions to be classified as efficient. This errs on the side of capturing *all* efficient transcodings at the cost of including some borderline inefficient ones.

In order to measure the overall efficiency of the transcoding algorithm, we performed experiments on all the images in our workloads. We measured the original JPEG Quality Factor of the images using the algorithm described earlier, reduced the image qualities to 75%, 50% and 25% of the original Quality Factor values, using fixed Huffman tables[1], and then measured the resulting image sizes. The change in image size, from the original image size, for a given reduction in image qual-

[1] It has been observed that using custom Huffman tables may yield better compression. For the images in our workload, we empirically measured that, on average, using custom tables produces images that are 11% smaller than using fixed tables. However, this comes at a cost of consuming 32% more memory and taking 56% more time to transcode. Our use of fixed tables means even more savings are possible.

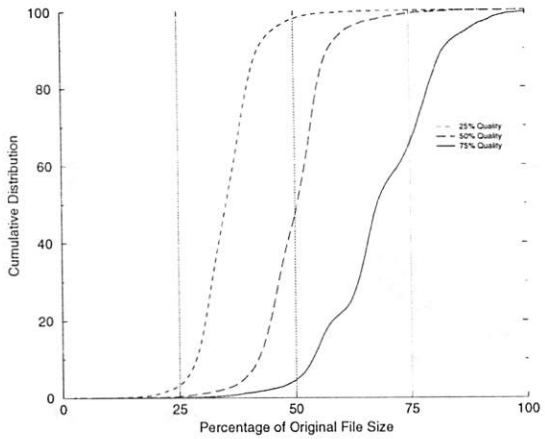


Figure 3: Cumulative Distribution of transcoding efficiency for all the images in the workloads

ity, is plotted as an cumulative distribution in Figure 3. For a given reduction in quality, any change in file size that is less than the change in quality (values to the left of the vertical line representing the loss in quality) is an efficient transcoding as the resulting file size is smaller than the corresponding loss in quality. Any value to the right is inefficient as the resulting file size is bigger than the corresponding loss in quality.

From Figure 3, we can see that reducing the initial quality to 75%, 50% and 25% was efficient for 66.1%, 47.4% and 3.2% of the images respectively. The results for the different workloads, were tabulated in Table 2.

From Table 2, we can see that images reduce efficiently for smaller drops in quality; a big drop in image quality to 25% of original quality does not lead to an efficient transcoding for most of the images. For images from PhotoNet, only 25% of the images reduced efficiently for a Quality Factor reduction to 75%, while 66% of the images reduced efficiently for a Quality Factor reduction to 50%.

In exploring this behavior, we discovered that images in the PhotoNet collection were initially entropy encoded using custom Huffman tables. Hence, for a small drop in image quality, the relative space gain from transcoding was offset by the increase in image size from using (less optimal) standard Huffman tables. For a more significant loss in the Quality Factor, the space gain from transcoding offsets this increase in image size.

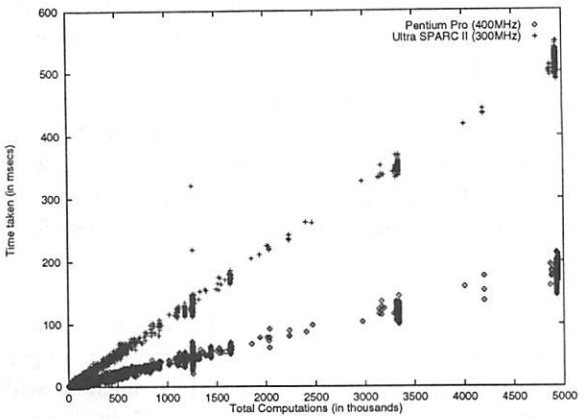


Figure 4: Computational Cost to re-quantize the JPEG images

4.2 Computation Cost

The next measure of the effectiveness of a transcoding is an analysis of the computational cost involved in performing a transcoding operation so that an intelligent decision can be made about the applicability of the transcoding for a particular operating environment.

The JPEG Quality Factor value of an image can be reduced without completely uncompressing the image (Table 1), by first performing an entropy decoding, followed by dequantization and requantization followed by entropy encoding. The computational cost does not depend on the quality of the image being transcoded, nor on the quality value being used for transcoding the image. Quantization and dequantization involve integer multiplication and division operation on each DCT image block, which can be collapsed into a single integer multiplication on each DCT image block. The basic computational block is of the form

$$dst[i] = (src[i] * reQuantizedScale) \gg scale\ (0 < i < 63)$$

For an image with i color components, and a size of $n_i \times m_i$ blocks per color component, we need to perform $\sum_i n_i * m_i * 64$ computations. For example, transcoding a 480×480 full color image (with 3 color components YCbCr) with no component scaling involves $60*60*3*64$ or 691200 computations. For a 480×480 image with 2×1 scaling for chrominance values, this corresponds to $(60 * 60 + 60 * 30 + 60 * 30) * 64$ or 460800 computations. Transcoding a progressive JPEG image requires expensive Huffman encoding for re-compression as the default tables are not valid for progressive formats. Hence, for our experiments, we automatically transcode progressive JPEG images to the simpler sequential format.

The image component dimensions are available in the JFIF headers and hence the number of basic computational blocks required to transcode an image can be computed for the particular image. Since the time taken to compute the basic computational block can be statically determined for a particular computing device, we want to validate that we can use the number of basic computations as a measure of the computational cost for a particular transcoding.

For our experiments, we used the IJG software, which is not particularly optimized for the compression operation. We used a Pentium II 400MHz machine with 128MB memory and an Ultra Sparc II 300MHz machine with 512MB of memory, both running Solaris 2.6 for our experiments. Based on the basic computational blocks described above, the time taken to transcode the images from our workloads are plotted in Figure 4. We observe that the number of basic computation blocks and the time to transcode an image are linearly correlated. The results indicate that the observed data fits the linear equation $time = const * basicBlks$ (where $const \simeq 37$, for Pentium II and $const \simeq 105$, for Ultra Sparc II). The value $time$ is measured in msecs and $basicBlks$ are measured in millions of basic computational blocks. The linear correlation coefficient (ρ) between the computational blocks and the time taken to transcode was 0.99 for both the machines. The 95% confidence interval, computed using standard deviation about regression, was measured at 15.3 msecs for the Ultra Sparc II and 7.9 msecs for the Pentium Pro II. These confidence intervals are quite adequate for most purposes.

Hence we conclude that the time required to perform a image transcoding using the JPEG Quality metric can be predicted accurately by computing the number of basic computational blocks using the image components dimensions. The time taken to compute a basic computational block, which affects the coefficients of the linear equation described above, can be computed statically for a particular computing device.

4.3 Predictable Size Tradeoff

Another objective in using JPEG Quality Factor as a transcoding algorithm is the ability to predict the output size of an image for a particular transcoding.

Previous work by Han et al. [11] attempted to predict the image size using the image pixel counts as a metric. They transcoded sample images to JPEG Quality Factor values of 5 and 50. For these transcodings, they measured the linear correlation coefficient between the transcoded output JPEG image size and the number of pixels in the image at 0.94 and 0.92 respectively. For the images in our data set, we plotted (Figure 5) a scatter plot of the input pixels vs. the output image size. The data exhibits heteroscedasticity [14] of the variance, wherein the variance of the output size is not the same for all values of input pixels. The variance is higher for images with higher input pixels. The studentized residual for transcoding images to a Quality Factor value of 5, with the i^{th}observation deleted, were plotted as a scatter plot in Figure 6. Similar results were observed for transcoding images to Quality Factor value of 50. In order to make inferences from the data sample using linear statistical models, the plot should contain a horizontal band of points with no hint of any systematic trend. The residual values for our data, as seen in Figure 6, suggests that the output size depends on multiple (unknown) parameters and not just on the input pixels. Hence the input pixels cannot be used as a useful predictor of the output image size for our set of images.

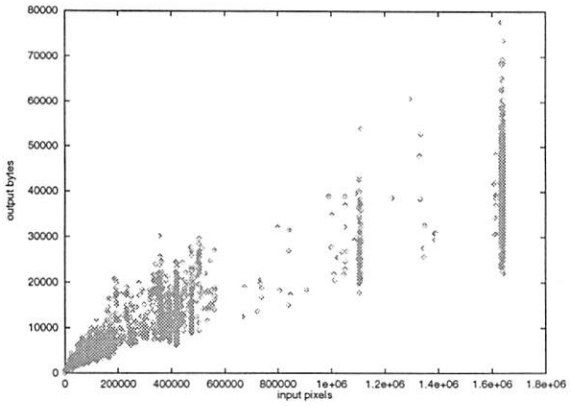


Figure 5: Correlation between output size and input pixels (Q=5)

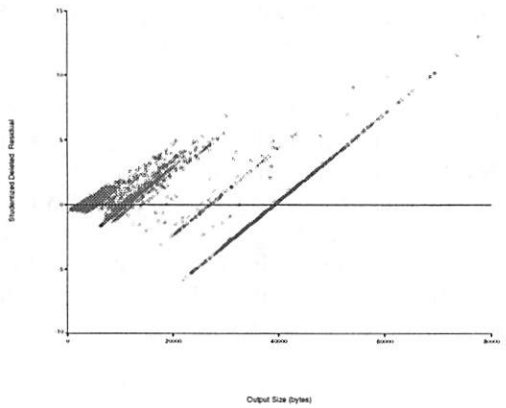


Figure 6: Studentized Residuals with i^{th}observation deleted (Q=5)

Intuitively, this makes sense as the type of input images can vary from photographic images to clip arts to images

of random noise. Since JPEG encoding is optimized for realistic images, random noise will not be encoded as efficiently as a photographic image. There is no easy heuristic to identify if an image is a photographic image. It is unlikely that the exact output size of an image for a particular transcoding would be predictable a priori without an understanding of what the image represents.

Even though the ability to predict the exact output image size would be valuable, for a transcoding to be useful, it is usually sufficient to analyze if an image will transcode efficiently. Hence, we try heuristics to predict if an image will transcode efficiently.

To better understand what factors might affect the transcoding efficiency, we need to understand the characteristics of the JPEG entropy coding algorithm. Since re-quantization only involves re-encoding of entropy parameters, DCT operations do not play a part in affecting the efficiency of a transcoding that lowers the JPEG Quality Factor.

JPEG baseline algorithm computes the entropy encoding using Huffman encoding with a standard Huffman table. The JPEG quantization step strives to increase the number of zeros in the image, starting with the least perceptible high frequency components. The default Huffman table stores zero value using 2 bits. Since typical images that are to be entropy encoded tend to be predominantly zero, the JPEG algorithm employs three additional optimization steps to improve the compression ratios.

1. The DC components are stored as a difference value from the DC coefficient from the previous image block, rather than by a straight Huffman code.
2. Sequences of zeros are encoded by a single byte which contains the count of zeros in the upper four bits with the lower four bits being zeros. For example, a sequence of 16 zeros are stored using 8 bits, instead of 32 bits required by pure Huffman encoding.
3. If the trailing entries in a image block are zeroes, the JPEG algorithm stores an End of Block (EOB) marker and avoids encoding those trailing zeros explicitly. Hence, if an image block had 60 trailing zeros, they will be encoded using 0 bits, instead of 120 bits that would be required by a pure Huffman encoding.

Empirically, we found that we only need to analyze the luminance values of an image to predict its transcoding efficiency. This makes sense, since human eyes are less sensitive to chrominance values; the default quantization tables reduce the chrominance values to zeros faster. Also, empirically, we found that images that typically transcode efficiently tend to have higher coefficients for the low frequency components. Since the high frequency components are reduced quicker by the quantization tables, their contribution does not affect the final image size. We also noticed that images of higher initial quality tended to transcode efficiently.

Prediction Criteria	% for Criteria	75% Q	50% Q	25% Q	Q= 75	Q= 50	Q= 25
None	*100*	*67*	*46*	*2*	*89*	*81*	*41*
$P = 0$	21	57	37	1	89	78	44
$P > 1$	29	85	58	6	86	81	30
$P > 3$	15	97	82	8	87	81	27
$P > 5$	9	99	83	12	89	84	30
$Q > 80$	20	98	83	9	87	86	30
$R < 10\%$	54	68	51	4	89	81	40
(Q > 80) or (P > 3)	21	96	82	9	87	85	30

Table 3: % of Accurately Predicted Images (C%)

Prediction Criteria	% for Criteria	75% Q	50% Q	25% Q	Q= 75	Q= 50	Q= 25
None	*100*	*1*	*6*	*82*	*8*	*9*	*38*
$P = 0$	21	0	2	18	2	2	8
$P > 1$	29	0	2	23	3	3	13
$P > 3$	15	0	0	12	2	2	6
$P > 5$	9	0	0	7	1	1	4
$Q > 80$	20	0	0	14	2	2	8
$R < 10\%$	54	0	2	43	5	5	20
((Q > 80) or (P > 3)	21	0	0	14	2	2	8

Table 4: % of Egregiously Mispredicted Images (E%)

Taking these observations about JPEG images as well as the Huffman optimizations into account, we hypothesize that an image will transcode efficiently, if:

There are sufficiently many high magnitude low frequency components in the luminance blocks of the images. The alternative, which are images whose magnitude of low frequency components are low, will already have lost much of the quality represented in the low frequency components that are most noticeable to the human eye and hence further compression is inefficient.

We define low frequency factor (P) as the percentage of low frequency components in the luminance blocks of an image that have coefficients whose absolute value is over a certain threshold. The number of frequency components to analyze is a compromise between choosing predominantly low frequency component images such as cartoons and predominantly high frequency component images such as detailed photographs. We experimentally varied the number of low frequency components to consider at 3, 5 and 7 (odd number of components

are required to maintain equal weighting for horizontal and vertical frequency components) and the magnitude thresholds to 64, 80, 96 and 128. We found that the values of 80 for the threshold and 5 for the number of components reasonably captured our intuition. Hence, for our experiments, we computed P as the percentage of frequency components 1 through 5 (which translates to 1, 8, 16, 9 and 2 in the natural order) over all luminance blocks that have coefficients with an absolute value greater than 80.

There are sufficiently few luminance color blocks within the image in which all of the highest frequency components beyond some point have coefficients that are all zero. Such images still present adequate opportunity for further efficient compression.The alternative, which is an image with many such blocks having zero coefficients for all of the high frequency components, will already be efficiently compressed using the JPEG Huffman optimizations described earlier.

We define a high frequency factor metric (R) as the percentage of luminance blocks within the image which have zero coefficients for *all* of the highest frequency components beyond a certain threshold. The choice of the threshold is a compromise. We tried zero-coefficient runs of the 48, 63, and 58 highest frequency components on test images. For our study, we chose 58 ($max - P or 63 - 5$) as the threshold, in order to avoid counting coefficients already captured by the low frequency factor (P), described above. Hence, for our experiments, we computed R as the percentage of luminance image blocks which have zero coefficients for all of the 58 (or more) highest frequency components.

The image is of high initial quality. We consider images produced with a JPEG Quality Factor value (Q) over 80 as high quality.

The parameters P and R can be computed easily by traversing a JPEG image's luminance DCT component blocks once. P can be computed by accessing the first few components in each DCT component block. Usually, these parameters can be computed as a by-product of transcoding an image locally or while measuring the initial JPEG Quality Factor of an image.

Using these parameters, we categorized the images and measured the number of images that transcoded efficiently in each category. For a particular criteria to be useful in predicting if an image will transcode efficiently, it needs to:

Apply to a significant percentage of images. For example, it is undesirable to have a prediction algorithm that makes predictions for only about 2% of the images. For our study, we analyze prediction criteria that at least predict 10% of images in a workload.

Accurately predict at a higher percentage than the general population. The percentage of images that are predicted efficient should be better than the percentage of efficient images in the original workload. For example, if 25% of all the images transcoded efficiently, we prefer a prediction that categorizes the images such that 80% of the predicted images transcode efficiently over a criterion that categorizes 30% of the efficient images.

Mis-predict egregiously inefficient images at a lower percentage than the general population. We define a pathological image as an image that is bigger than 125% in size than the loss in information quality. For example, if an image loses 50% Quality, we categorize an image that is bigger than 125% of 50%, i.e. 62.5%, as pathological.

We first transcoded the images in our workload to Quality Factor values of 75%, 50% and 25% of the original Quality Factor values as well absolute Quality Factor values of 75, 50 and 25. Using various predictor criteria values, we separated the images from our workload into images that were predicted as efficient images by the particular criteria. For the images that were predicted efficient, we measured the percentage of images that actually transcoded efficiently as well as the percentage of pathological images that were mispredicted as efficient. The goal was to increase the percentage of good predictions. The percentage of correct predictions (C%) and the percentage of egregious mispredictions (E%) were tabulated in Table 3 and 4 respectively. The results were tabulated as follows:

The first column in the table shows the criteria that were used to separate the images. We call the criterion *None*, which includes all the images in the workload as the base case.

The second column specifies the percentage of images that satisfy the criteria specified in the first column. For our study, values less than 10% were ignored because they categorize an insignificantly small percentage of images to be useful as a prediction criterion.

The subsequent columns show the results for transcoding the images to 75%, 50% and 25% of the original JPEG Quality Factor as well as absolute JPEG Quality Factor values of 75, 50 and 25.

The goal is to choose results that produce significantly higher C% and lower E% than the base case. The results for all the images in our workload are shown in Table 3 and 4 respectively. Similar trends were observed for the individual workloads.

From Tables 3 and 4, we note that $P > 5\%$ predicts less than 10% of the images (9%) and hence is ignored from further consideration. For transcoding to Quality Factor values of 75%, 50% and 25% of the original JPEG Quality Factor, the criterion $(Q > 80) \,||\, (P > 3\%)$ predicts efficient images correctly and mispredicts images at a rate better than the base case. For transcoding to JPEG Quality Factor values of 75, 50 and 25, none of the criteria predicts at a significant rate, even though they mispredict images at a better rate than the base case. The value of R is not a useful predictor of the efficacy of an image for transcoding using the JPEG compression metric.

We cannot make conclusions regarding images that weren't predicted as efficient. We tried parameters such as number of colors, pixel density etc, but could not find a reliable way of predicting the efficiency of images that have $P < 3$ and $Q < 80$. We repeated the experiments by varying the definition of P to include images frequency components over absolute values of 64, 96 and 128 as well as changing the number of frequency components to 3 and 7. The end results were similar and the values that we chose better predict efficient transcodings by a slight margin.

5 Related Work

Fox et al.[8] used transcoding to render an image on a PDA such as Palmpilot, as well as to offset access latencies from slow modems. Noble et al. [18] manipulated the JPEG Compression metric as a distillation technique for a web browser that adapts to changing network environments. Mazer et al. [16] describe a framework for allowing users to specify their own transcoding transducers for a application-specific proxy that acts on the HTTP stream to customize the stream for a particular client. Ortega et al. [19] have used JPEG progressive encoding to recode images to lower resolutions, thereby increasing the effective cache size.

Commercial products such as WebExpress [6] from IBM, QuickWeb technology [12] from Intel, Fastlane [21] from Spectrum Information technology and Johnson Grace ART format [1] from AOL have used various forms of compression and transcoding operations to improve web access from slow networks.

Even though transcoding has been widely used in a number of systems to deal with network access latencies, display characteristics or storage space requirements, there has been little formal work in conducting a systematic study to measure the information loss associated with a given transcoding. As previously mentioned, Han et al.[11] attempted a similar effort to characterize image transcoding delay as well as the transcoded image size.

The concept of information quality and measuring the objective and subjective quality of an image has been well researched and understood. The ability to quantify the information loss in rendering an image on a particular hardware is central to measuring the performance of rendering devices such as monitor, plotters, printers etc. CIE publishes guidelines on measuring the color reproduction characteristics. The International Telecommunication Union (ITU) publishes recommendations for subjective assessment of quality of television pictures.

6 Summary

The ultimate goal of our work is to increase the effectiveness of the transcoding technique applied to internet data access. We currently focus on JPEG images. Given a particular image, we want to be able to determine whether performing a transcoding operation will pay off and how aggressively it should be done. Toward that end, this paper makes the following contributions to our understanding of the use of image transcoding:

1. We have defined the notion of a "quality aware transcoding" as a transformation that explicitly trades off image information quality for reductions in object size (e.g. transmission length or storage size).
2. We have found previous work that supports the use of the JPEG Quality Factor to capture our requirement for a quantifiable measure of image information quality. This allows us to consider JPEG compression metric as a candidate for quality aware transcoding.
3. We have developed an algorithm to estimate the JPEG Quality Factor used to originally produce an image. This is necessary to make "loss of quality" a meaningful concept.
4. We have characterized a sample of typical images available on the internet with respect to their origi-

nal size and JPEG Quality. This allows us to evaluate our work in the context of a realistic workload.

5. We have developed an algorithm to predict the computational cost involved in transcoding an image. This allows us to evaluate the cost penalty in performing a transcoding operation.

6. We have developed criteria that can be used to predict whether an image will transcode efficiently, avoiding serious mispredictions. This allows us to evaluate if it will be worthwhile to transcode a particular image so as to achieve a higher information quality per byte of image storage.

We are currently investigating heuristics to identify whether a transcoding from a GIF to a JPEG image format will be efficient for a particular image.

7 Acknowledgments

This research was supported in part by a dissertation research grant from IBM. We wish to thank Ashish Gehani for many discussions of the JPEG compression algorithm. We also thank Susan Paddock for pointing us to the correct statistical analysis techniques. We also thank our shepard, Prof. Eric Brewer for his valuable comments.

References

[1] America Online Inc. Johnson grace ART image format.

[2] D. Bourgin. Color spaces FAQ. www.well.com/user/rld/vidpage/color_faq.html, May 1995.

[3] CCITT Recommendation T.81, International Telecommunication Union(ITU), Geneva. *Digital Compression and Coding of Continous-Tone Still Images - Requirements and guidelines*, Sep 1992.

[4] S. Chandra, C. S. Ellis, and A. Vahdat. Multimedia Web Services for Mobile Clients Using Quality Aware Transcoding. In *The Second ACM International Workshop on Wireless Mobile Multimedia*, Seattle, Aug 1999.

[5] CNN Interactive: All Politics. cnn.com/ALLPOLITICS/, Dec 1998.

[6] R. Floyd, B. Housel, and C. Tait. Mobile Web Access using eNetwork Web Express. *IEEE Personal Communications*, 5(5), Oct 1998.

[7] A. M. Ford. *Relations between Image Quality and Still Image Compression*. PhD thesis, University of Westminster, May 1997.

[8] A. Fox and E. A. Brewer. Reducing www latency and bandwidth requirements via real-time distillation. In *Proceedings of Fifth International World Wide Web Conference*, pages 1445–1456, Paris, France, May 1996.

[9] P. Greenspun. www.photo.net, Dec 1998.

[10] E. Hamilton. *JPEG File Interchange Format - Version 1.02*. C-Cube Microsystems, Sep 1992.

[11] R. Han, P. Bhagwat, R. LaMaire, T. Mummert, V. Perret, and J. Rubas. Dynamic adaptation in an image transcoding proxy for mobile web browsing. *IEEE Personal Communications Magazine*, 5(6):8–17, Dec 1998.

[12] Intel Quickweb. intel.com/quickweb.

[13] A. A. Jr. and C. H. Null. Image quality: a multidimensional problem. In A. B. Watson, editor, *Digital Images and Human Vision*, pages 141–148. MIT Press, Cambridge, MA, Nov 1993.

[14] D. G. Kleinbaum, L. L. Kupper, and K. E. Muller. *Applied Regression Analysis and Other Multivariable Methods*. PWS - Kent Publishing Company, second edition, 1988.

[15] T. Lane, P. Gladstone, L. Ortiz, J. Boucher, L. Crocker, J. Minguillon, G. Phillips, D. Rossi, and G. Weijers. The independent jpeg group's jpeg software release 6b. ftp.uu.net/graphics/jpeg/jpegsrc.v6b.tar.gz.

[16] M. S. Mazer, C. Brooks, J. LoVerso, L. Theran, F. Hirsch, S. Macrakis, S. Shapiro, and D. Rockwell. Distributed clients for enhanced usability, reliability, and adaptability in accessing the national information environment. Technical report, The Open Group Research Institute, Cambridge MA 02142, 1998.

[17] Media Graphics International, Arvada, CO. 70,000 Multimedia Graphics Pack, 1997.

[18] B. D. Noble, M. Satyanarayanan, D. Narayanan, J. E. Tilton, J. Flinn, and K. R. Walker. Application-aware adaptation for mobility. In *Proceedings of the 16th ACM Symposium on Operating Systems and Principles*, Saint-Malo, France, Oct 1997.

[19] A. Ortega, F. Carignano, S. Ayer, and M. Vetterli. Soft caching: Web cache management techniques for images. In *IEEE Signal Processing Society 1997 Workshop on Multimedia Signal Processing*, Princeton NJ, Jun 1997.

[20] W. B. Pennebaker and J. L. Mitchell. *JPEG - Still Image Data Compression Standard*. Van Nostrand ReinHold, NY, 1993.

[21] Spectrum Information Technologies Inc. Fastlane. spectruminfo.com.

[22] Star Wars - Episode I. starwars.com/episode-i/, Dec 1998.

Secondary Storage Management for Web Proxies

Evangelos P. Markatos Manolis G.H. Katevenis
Dionisis Pnevmatikatos Michail Flouris*

Computer Architecture and VLSI Systems Division
Institute of Computer Science (ICS)
Foundation for Research & Technology – Hellas (FORTH)
P.O.Box 1385, Heraklio, Crete, GR-711-10 GREECE
http://archvlsi.ics.forth.gr markatos@csi.forth.gr

Abstract

World-Wide Web proxies are being increasingly used to provide Internet access to users behind a firewall and to reduce wide-area network traffic. Recent results suggest that disk I/O is increasingly becoming the limiting factor for the performance of web proxies. In this paper we study the overheads associated with disk I/O for web proxies, and propose secondary storage management alternatives that improve performance. We use a combination of experimental evaluation and simulation based on traces from busy web proxies. We show that web proxies experience significant overheads due to disk I/O. We propose several file management methods that reduce the disk I/O overhead overhead by a factor of 25 overall, resulting in a single-disk service rate that exceeds 500 (URL-get) operations per second.

1 Introduction

World-Wide Web proxies are being increasingly used to provide Internet access to users behind a firewall and to reduce wide-area network traffic. Recent results suggest that disk I/O overhead is becoming an increasingly important bottleneck for the performance of web proxies. For example, Rousskov and Soloviev [33] observed that disk delays contribute about 30% toward total hit response time. Mogul states that their observations of the web proxy at Digital Palo Alto firewall suggest the disk I/O overhead of caching turns out to be much higher than the latency improvement from cache hits [27]. Thus, to save the disk I/O overhead the server is typically run in its non-caching mode.

In this paper we study the overheads associated with disk I/O for web proxies, and propose secondary storage management alternatives that improve performance. We show that the single most important source of overhead is associated with storing each URL in a separate file. File system operations (like a file creation followed by a file deletion) may easily take up to 50 milliseconds (aggregate), even on modern hardware. Given that the median size of a cached file is 3KBytes [33], and that for each URL typical web proxies create one file (to store the contents of the URL) and delete another file (to free space), then the rate at which a web proxy can store data to disk is 3KBytes every 50 msec, or roughly 60 KBytes/sec, a rate of execution that is orders of magnitude lower than the data transfer rates that current disks can sustain. This data rate is even lower than most Internet connections. To alleviate this file management overhead we propose a storage management method called BUDDY that stores several URLs per file. BUDDY identifies URLs of similar sizes ("buddies") and stores them in the same file. BUDDY reduces file management overhead by storing several URLs per file and reduces fragmentation by storing similar-sized URLs in each file.

Once we reduce file management overhead using BUDDY, we show that the next largest source of overhead is associated with disk head movements due to file write requests which write data in widely scattered places over the disk space. To improve write throughput, we propose a file space allocation algorithm (called STREAM) inspired from log-structured file systems [32]. STREAM stores all URLs in a single file contiguously (if possible). STREAM reduces disk seek and rotational overheads and manages to perform write operations at maximum speed. Once write operations proceed at maximum speed, URL read operations emerge as the next largest source of overhead. To reduce the disk read overhead we propose algorithms that cluster several read operations together (LAZY-READS) (in order to reduce the disk head seek time), and organize the layout of the URLs on the file so that URLs accessed together are

*All the authors are also with the Unversity of Crete.

stored in nearby file locations (locality buffers).

To evaluate the performance of our approach we use a mix of trace-driven simulation and experimental evaluation. Traces from the DEC's web proxy are fed into a 512-Mbyte main memory LRU cache simulator [7]. URLs that miss the main memory cache are fed into a 2-Gbyte disk LRU cache simulator. URLs that miss this second-level cache are assumed to be fetched from the Internet. These misses generate `URL-write` requests, because once they fetch the URL from the Internet they save it on the disk. Second-level URL hits generate `URL-read` requests, since they read the contents of the URL from the disk. To make space for the newly arrived URLs, the LRU replacement policy deletes non-recently accessed URLs resulting in `URL-delete` requests. All `URL-write`, `URL-read`, and `URL-delete` requests are fed into a file space simulator which maps URLs into files (or blocks within files) and sends the appropriate calls to the file system. Our results suggest that BUDDY achieves an order of magnitude improvement over traditional web proxy approaches, STREAM achieves a factor of 2-3 improvement over BUDDY, and locality buffers achieves a 20%-150% improvement over STREAM.

The rest of the paper is organized as follows: Section 2 surveys previous work. Section 3 presents our algorithms, and evaluates their performance. Section 4 summarizes and concludes the paper.

2 Previous Work

Caching is being extensively used on the web. Most web browsers cache documents in main memory or in local disk. Although this is the most widely used form of web caching, it is the least effective, since it rarely results in large hit rates [1]. To improve cache hit rates, caching proxies are used [8, 39]. Proxies employ large caches which they use to serve stream of requests coming from a large number of users. Since even large caches may eventually fill up, cache replacement policies have been the subject of recent research [1, 7, 20, 23, 31, 34, 40, 41]. Sophisticated caching mechanisms usually improve the observed user latency and reduce network traffic. Some caches may even employ intelligent *prefetching* methods to improve the hit rate even further [3, 4, 13, 24, 38, 29, 37].

Recently, it was realized that web proxies spend a significant percentage of their time doing disk I/O. Rousskov and Soloviev observed that disk delays contribute 30% towards total hit response time [33]. Mogul suggests that disk I/O overhead of disk caching turns out to be much higher than the latency improvement from cache hits [27]. To reduce disk I/O overhead Soloviev and Yahin suggest that proxies should have several disks, and that each disk should have several partitions. Using sophisticated write distribution policies Soloviev and Yahin are able to spread requests over several disks and to cluster requests on the same partition to avoid long seek delays [36]. Scott Fritchie found that USENET News servers spend a significant amount of time storing articles in files "one file per article" [12]. To reduce this overhead he proposes to store several articles per file and to manage each file as a cyclic buffer. His implementation shows that storing several URLs per file results in significant performance improvement. Maltzahn, Richardson and Grunwald [21] measured the performance of web proxies. With regard to disk I/O they measured that several disk accesses are needed for each URL request (in the average). This implies that the disk subsystem is required to perform a large number of requests for each URL accessed and thus it can easily become the bottleneck. In their subsequent work, they propose two methods to reduce disk I/O for web proxies [22]:

- they preserve locality of the http reference stream by storing files of the same web server in the same proxy directory (SQUIDL) and
- they use a single file to store all objects less than 8K in size (SQUIDM).

It seems that both the authors of this paper as well as Maltzahn, Richardson and Grunwald have independently discovered similar key ideas that reduce the disk overhead of a web proxy. For example, SQUIDL and SQUIDM of [22] use similar locality and file management principles to our algorithms (MULTIPLE-DIRS and BUDDY respectively). These common ideas reduce the file management (meta-data) overhead associated with storing URLs in a file system. However, our work also presents a clear contribution towards improving data (not meta-data) access overhead:

- We propose and evaluate STREAM and STREAM-PACK, two file-space management algorithms that (much like log-structured file systems) optimize write performance by writing data contiguously on the disk.
- We propose and evaluate LAZY-READS and LAZY-READS-LOC, two methods that reduce disk seek overhead associated with read (URL hit) operations.

As a result our algorithms improve on the performance of SQUID by a factor more than 25, while [22] reports performance improvements of a factor close to 5.

Most web proxies have been implemented as user-level processes on top of commodity (albeit state-of-the-art) file-systems. Some other web proxies were built on top of custom-made file systems or operating systems. NetCache was build on top of WAFL, a file system that improves the performance of write operations [16]. Inktomi's traffic server uses UNIX raw devices [18]. CacheFlow has developed CacheOS, a special-purpose operating system for proxies [17]. Unfortunately very little information has been published about the details and performance of such custom-made web proxies, and thus a direct quantitative comparison between our approach and theirs is very difficult. Although custom-made operating systems and file-systems can result in the best performance, we chose to explore the approach of running a web proxy as a user-application on top of a commodity operating system. Our approach will result in higher portability and more widespread use of web proxies.

The main contributions of our work in the area of disk I/O of web proxies are:

- We identify as the single largest source of overhead the storage of each URL in a separate file. We show the extend of this overhead, and propose a novel file management algorithm (BUDDY) to reduce it by an order of magnitude.
- We identify as the next single largest source of overhead the cost associated with file write operations. We propose a file space management approach (inspired by log-structured file systems) called STREAM that groups several independent write requests into long sequential writes that minimize disk head movement.
- Once write operations proceed at maximum speed (with the use of STREAM-based algorithms), read operations (although fewer in number) represent the next single largest source of overhead. We propose novel methods (LAZY-READS and LAZY-READS-LOC) that reduce the disk head movements associated with disk read operations.

3 Evaluation

3.1 Methodology

To evaluate the disk I/O performance of web proxies we use a combination of simulation and experimental evaluation as shown in Figure 1. We use traces from a SQUID web proxy used at Digital Equipment Corporation (`ftp://ftp.digital.com/pub/DEC/traces/proxy/webtraces.html`). We feed these traces to a *first-level* main memory cache simulator [7]. The simulated main-memory is 512 Mbytes large and replaces URLs using the Least-Recently Used (LRU) replacement policy. [1] URL requests that miss the main memory cache are fed into a *second-level* cache simulator that simulates the magnetic disk cache. Second-level hits read the contents of the URL from the disk and generate a *URL-read* request. Second-level misses are assumed to be sent to the appropriate web server over the Internet, and the server's response is saved in the disk generating a *URL-write* request. When the disk runs out of space, an LRU replacement algorithm is invoked, which may delete old files generating a *URL-delete* request. URL-delete requests are also generated when new versions of cached files are requested. The generated URL-read, URL-delete, and URL-write requests are sent to a file-space management simulator which forwards them to a Solaris UFS file system which reads, deletes, and writes URLs as requested. In all our experiments we report the total time (completion time) to serve the first million URL-read/URL-write/URL-delete operations. [2] The completion time reported in our experiments is inversely proportional to the throughput (operations per second) of the system and thus is a direct measure of it. If for example, the completion time reported is 2000 seconds, then the throughput of the system is 1058206/2000=529 URL-get requests per second. It is possible to argue however, that, besides throughput, latency is also an important metric, especially for the end user. However, latency (by itself) can be a misleading performance metric for our work. For example, suppose that proxy server A has a 15 msec average operation response latency and manages to sustain 50 operations per second, while server B has a 30 msec average operation response latency and manages to sustain 100 operations per second. Although latency may favor server A, most implementations will probably prefer to use server B, since it achieves higher throughput and its increase in latency is not noticeable by most humans. For this reason, our performance results focus on server throughput while making sure that our policies do not increase latency noticeably. This happens in most cases without particular effort because our policies interact with disks that operate in the millisecond range, while the typical world-wide web latency is in the second range (3-4 seconds per request in the average when lightly loaded, and more than 10 seconds per request when heavily loaded [2, 19]).

[1] Although more sophisticated policies than LRU have been proposed they do not influence our results significantly.

[2] The file space management simulator is fed with 1058206 URL-get requests that generate one million URL-read/URL-write/URL-delete operations. These 1058206 URL-get requests result in 338081 (32%) main memory hits, 42085 (4%) secondary memory hits, and 678040 (64%) misses, which result in 678040 URL-write operations, 42085 URL-read operations, and 279875 URL-delete operations.

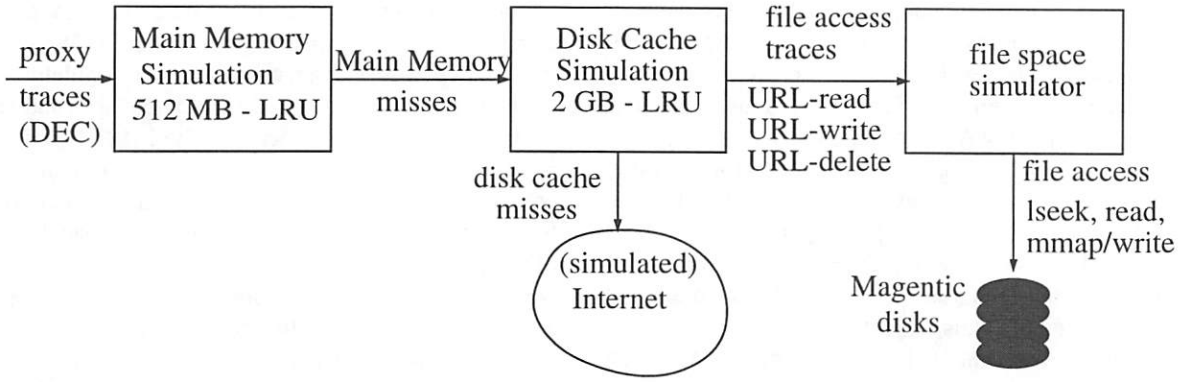


Figure 1: **Simulation Methodology.** Traces from the DEC's web proxy are fed into a 512-Mbyte main memory LRU cache simulator. URLs that miss the main memory cache are fed into a 2-Gbyte disk LRU cache simulator. URLs that miss this second-level cache are assumed to be fetched from the Internet. These misses generate `URL-write` requests because once they fetch the URL from the Internet they save it on the disk. Second-level URL hits generate `URL-read` requests, since they read the contents of the URL from the disk. To make space for the newly arrived URLs, the LRU replacement policy deletes non-recently accessed URLs resulting in `URL-delete` requests. All `URL-write`, `URL-read`, and `URL-delete` requests are fed into a file space simulator which maps URLs into files (or blocks within files) and sends the appropriate calls to the file system.

Our experimental environment consists of an ULTRA-1 workstation running Solaris 5.6, equipped with a Seagate ST15150WC 4Gbyte disk with 9 ms average latency, 7200 rotations per minute, on which we measured a maximum of 4.7 Mbytes per second write throughput.

3.2 Workload Characterization

In this first set of experiments we will demonstrate that the traffic sent to the disk subsystem of a web proxy is dominated by write requests. Figure 2 plots the number of URL-read and URL-write operations that are sent to the file system of the proxy server (for 5 million URL-get requests). The number of URL-write operations is around 3 million, and decreases slowly with disk size (since larger disks imply fewer URL misses). The number of URL-read requests is less than half a million and increases with disk size (since larger disks imply more URL hits). [3]

Figure 2 suggests that the number of URL-read operations is significantly smaller than the number of URL-write operations. This is because URL-write operations correspond to second-level URL misses which can be quite large, while URL-read operations correspond to URLs that miss the first-level cache but hit in the second-level cache, which are usually a small percentage.

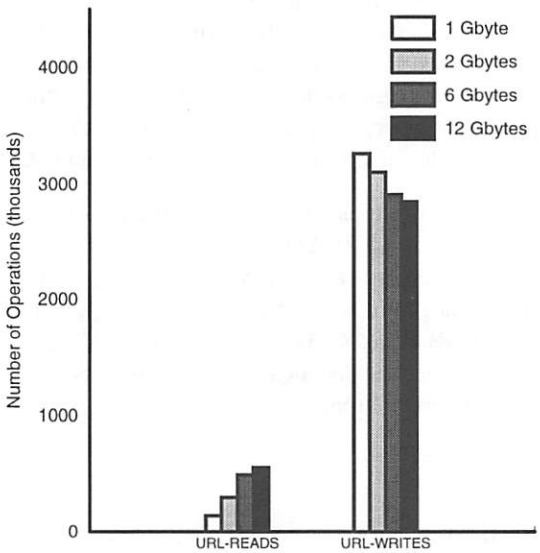


Figure 2: **File System Operations.** The figure displays the amount of URL operations (read/write) during the execution of a web proxy for various disk sizes. In all cases, write operations outnumber read operations.

[3] Note that the sum of URL-read and URL-write requests is in all cases 3.4 million and not 5 million as one might expect. This because the 512-Mbyte first level cache is able to achieve a 32% URL hit rate, which leaves 3.4 million URL requests for the second level cache.

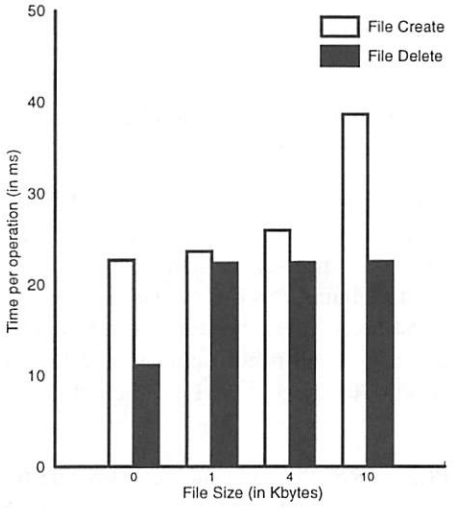


Figure 3: **File management Overhead.** The figure displays the cost of file creation and file deletion as measured by the HBENCH-OS (latfs). The benchmark creates 64 files and then deletes them in the order of creation. The same process is repeated for files of varying sizes. We see that the time to create a file is more than 20 msec. The time to delete a file is between 10 and 20 msec. The time to create and delete a 4-Kbyte file is close to 50 msec, which implies that this file system can create no more than 20 such files per second.

3.3 File Management Overhead

In this set of experiments we will demonstrate that the single most important overhead associated with the disk I/O of web proxies is the result of storing each URL in a separate file. To measure this file system overhead we use the HBENCH-OS benchmark [6] that creates 64 files in a directory and then deletes them in reverse-of-creation order. After the benchmark is repeated a number of times, the average time to do file operation, as well as its standard deviation are reported. In all but two cases the standard deviation was less than 1% away of the average time, and in the two remaining cases it was less than 10%. Figure 3 plots the results. We see that the time to create a file is more than 20 milliseconds - even when the file is empty. The time to create a 10-Kbyte-large file is close to 40 milliseconds. The time to delete a file is more than 20 milliseconds (for non-zero-sized files). If we use a benchmark that creates/deletes more than 64 files, these times will go up since the traditional UNIX directory lookup takes time linear in the directory length. Published research results using similar benchmarks agree with our measurements [26].

For each URL fetched from a web server, a typical web proxy needs to create a file to store the contents of the URL. When the disk subsystem runs out of space, for each new file created (in the average) an old file will have to be deleted (to make free space). Thus, for each URL fetched from a web server, one file is created and one file is deleted. Figure 3 suggests that the cost of a file creation and a file deletion is about 50 msec, which implies that a web proxy that incurs such a file creation/deletion cost can fetch from the network (and store in the local disk) no more than 20 URLs per second. Given that the median size of a cached file is only about 3 Kbytes long [33], then the web proxy can serve data at a rate of 3 Kbytes every 50 msec, or about 60 Kbytes per second, a throughput that is two orders of magnitude smaller than most modern magnetic disks provide. This throughput is even smaller than most Internet connections. Thus, it is obvious why researchers observe that " the disk I/O overhead of caching turns out to be much higher than the latency improvement from cache hits" [27].

3.4 File Management

Most publicly available popular web proxies (including Squid [39], Harvest [8], and CERN) store each URL on a separate file. These files are stored in a shallow directory hierarchy (like Squid) or in a deep directory hierarchy (like CERN and Harvest). We believe that file management can be the largest limiting factor in the performance of a web proxy. To alleviate this performance bottleneck we propose a novel file-grouping method called BUDDY. The main idea behind BUDDY is that each file may store several URLs. URLs that need one block of disk space (512 bytes) are all stored in the same file. URLs that need two blocks of disk space are stored in another file, etc. Each file essentially is composed of same-sized slots. Each new URL is stored is the first free slot of the appropriate file. BUDDY behaves as follows:

- BUDDY creates one file to store all URLs that are smaller than one block, another file to store all URLs that are larger than a block, but smaller than two, another file to store all URLs that are larger than two blocks, but smaller than three, and so on, up to a predetermined number of blocks (THRESHOLD). URLs larger than this number are stored in separate files - one URL per file.

- On a file-write request for a given size, BUDDY finds the first free slot on the appropriate file, and stores the contents of the new URL there. If the size of the contents of the URL is above a certain threshold (128 Kbytes in most of our experiments), BUDDY creates a new file to store this specific URL only. [4]

[4] The effect of this threshold on performance is studied in figure 5.

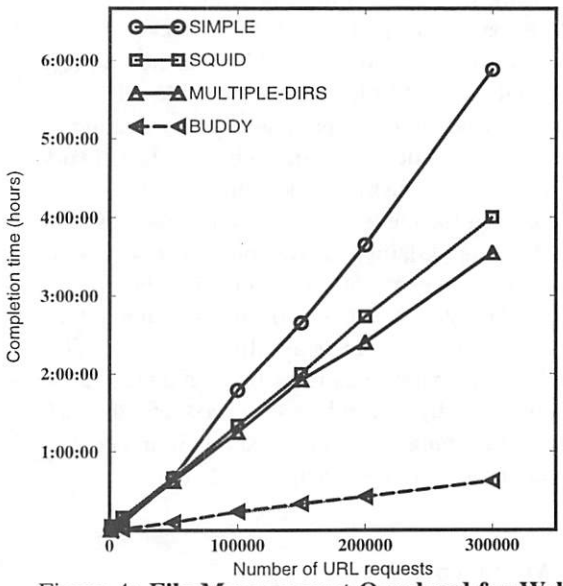


Figure 4: **File Management Overhead for Web Proxies.** The figure displays the overhead of doing 300,000 second-level cache file system operations on a 1-Gbyte disk. We see that both methods that create one file for each URL they need to store perform very bad. BUDDY, which stores several URLs per file takes roughly less than 9 msec per file operation.

- On a file-delete request, BUDDY marks the corresponding slot in the appropriate file as free. This slot will be reused at a later time by another URL of the given size.
- On a file-read request, BUDDY finds the slot in the appropriate file and reads the content of the requested URL.

The main advantage of BUDDY is that it practically eliminates the overhead of file creation/deletion operations. [5] The URLs that occupy a whole file of their own, represent a tiny percentage of the total number of URLs, so that their file creation/deletion overhead is not noticeable overall. One more advantage of BUDDY is that by placing same-sized URLs on the same file, it eliminates file space fragmentation; that is, a URL always occupies consecutive bytes within a file. This simplifies the mapping between URLs and the positions within files where they reside.

[5] BUDDY may also be used to reduce internal fragmentation and improve hit ratio by storing more data on a given disk. Current architecture trends suggest that disk block size should increase. This implies that small files, which occupy at least one disk block, in the future will probably occupy significantly more space than needed. On the contrary, when storing several URLs per file, as BUDDY does, internal fragmentation will be reduced, more data will fit into the disk, and higher hit rates will be possible.

To evaluate the performance advantages of BUDDY we compared it against three traditional approaches:

- SIMPLE: this approach stores each URL in a separate file. All files reside in the same directory.
- SQUID: this approach, used by the SQUID proxy server, creates a two-level directory structure. The first level contains 16 directories (named 0..F), while the second level contains 256 directories (named 00..FF) for each first-level directory. Files are written in the directories in a round robin manner: the first file is written at 0/00, the next at 0/01, ... then at 0/FF, then at 1/00, etc.
- MULTIPLE-DIRS: this approach creates one file for each URL. All files that correspond to URLs from the same server are stored in the same directory. Files that correspond to URLs from different servers are stored in different directories. All directories are in the same level.

Figure 4 shows the completion time of a stream of 300,000 file-system requests (URL-read, URL-write, URL-delete), which were generated by 398034 URL-get requests as a function of the management algorithm used. We see that SIMPLE has the worst performance, serving about 14 URL-get operations per second. SQUID performs better - it achieves 20 URL-get operations per second; independent published performance results also suggest that SQUID achieves 20-25 URL-get operations per second on a single-disk system [9]. MULTIPLE-DIRS performs a little better, achieving 23 URL-get operations per second.

Compared to SIMPLE, SQUID, and MULTIPLE-DIRS, BUDDY improves performance almost by an order of magnitude, since it achieves close to 133 URL-get operations per second. This is because BUDDY neither creates nor deletes files for most of the URLs it serves.

The careful reader however, will notice that SIMPLE, SQUID, and MULTIPLE-DIRS appear to be more robust than BUDDY in a case of system crash. If the system crashes, SIMPLE, SQUID, and MULTIPLE-DIRS will probably recover a large percentage of their metadata, while BUDDY will probably lose some portions of its metadata (i.e. where is each URL stored). We believe that this is not a significant problem for the following reasons:

- BUDDY can periodically (i.e. every few minutes) write its metadata information on safe storage, so that in the case of crash it will lose only the work of the last few minutes.

- Alternatively, BUDDY can store along with each URL, its name and size. In case of a crash, after the system reboots, the disk can be scanned, and the information about which URLs exist on the disk can be recovered.
- Even if few cached documents are lost due to a crash, they can be easily retrieved from the web server where they permanently reside. Thus, a system crash does not lose information *permanently*; it just loses the *local copy* of some data (i.e. a few minutes worth) which can be easily retrieved from the web again.
- There exists a significant amount of recent work that speeds-up synchronous disk write-operations (and thus metadata updates) by using for example Non-volatile RAM [42], transactions [14], replication [30], or soft-updates [25].

In BUDDY, URLs that are larger than a threshold are stored in a separate file each - one URL per file. All other URLs are "buddied" together in appropriate files. The next experiment sets out to explore how large this threshold should be. Figure 5 plots the completion time of the BUDDY as a function of the threshold. We see that as the threshold increases, the completion time of BUDDY improves fast. This is because an increasing number of URLs are stored in the same file, eliminating a significant number of file create/delete operations. As the threshold increases above 10 (disk blocks), the completion time improves, but not as fast. When the threshold reaches 256 blocks (i.e. 128 Kbytes), we get (almost) the best performance. Our results suggest that URLs larger than 128 Kbytes should be given a file of their own. Such URLs are rare and large, so that the file creation/deletion overhead is not noticeable.

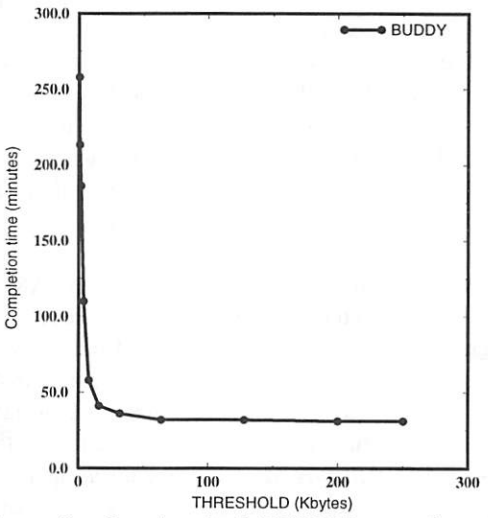


Figure 5: **Overhead of BUDDY as a function of the THRESHOLD.** The figure displays the cost of serving 400,000 file system operations as a function of the threshold used by BUDDY. The experiment suggests that URLs smaller then 128 Kbytes should be "buddied" together. URLs larger than 128 Kbytes can be safely given a file of their own (one URL per file) - they will not result in any noticeable overhead.

3.5 Optimizing Write throughput

Once we reduce the file management overhead we noticed that the next single largest source of overhead is due to disk latencies incurred by writing data scattered all over the disk. Although it reduces file management overhead, BUDDY makes no effort to layout data on disk in such as way as to improve write (and/or read) performance. Given that a web proxy's disk workload is write-dominated (as shown in figure 2), the performance of write operations can be improved if writes to the disk happen in a log-structured fashion. Thus, instead of writing new data in *some* free space on the disk, we continually *append* data to the disk until the disk runs out of space, in which case write operations continue from the beginning of the disk. This method has been widely used in log-structured file systems [5, 15, 28, 35]. In this paper we use a log-based approach in *user-space* management to see if the effectiveness of log-structured file system can be achieved by a user program (the web proxy) without the need of a specialized file system. Towards this end we develop a *file-space* management algorithm (called STREAM) that (much-like log-structured file systems) streams write operations to the disk:

- The web proxy stores *all* URLs in a single file organized in slots of 512 bytes long. Each URL occupies an integer number of such slots.
- URL-delete operations mark the appropriate space on the file as free.
- URL-read operations read the appropriate portions of the file that correspond to the given URL.
- URL write operations continue appending data to the file until the file runs out of space (i.e. they reach the end of file). In this case, new URL write operations continue from the beginning of the file writing on free slots, until they reach the end of file, etc.

STREAM has the potential of making long sequential write operations. The length of these sequential write operations depends on the distribution of the free space on the file, which in turn depends on the amount of scratch space that is available to the file. For example, if there is no scratch space,

then there will always be only one free slot in the file, which will tend to move non-sequentially in the file, and STREAM will have little opportunity to make long sequential writes. The whole purpose behind STREAM (and log-structured file systems) is that disks should be operated at much less than 100% of their utilization, so that there is always enough free space on the disk. This free space will be used to write new files/data in long sequential write operations.

When we first evaluated the performance of STREAM we noticed that even when there was always free space and even in the absence of read operations, STREAM did not write to disk at maximum throughput. We traced the problem and found that we were experiencing a *small-write* performance problem: writing a small amount of data to the file system, usually resulted in a *disk-read* and a *disk-write* operation. The reason is the following: if a process writes a small amount of data (e.g. the first block of a page) in a page that is not in the main memory cache, the operating system will *read* the page from the disk, make all updates in main memory, and then write the page to the disk. To reduce these small-write effects we developed a packetized version of STREAM: STREAM-PACKETIZER that works just like STREAM with the following exception:

- There exists a packetizer buffer that is one page long and aligned to a page boundary. URL-write operations are not being forwarded to the file system - instead they are being accumulated into a packetizer as long as they are stored contiguously to the previous URL-write request. Once the packetizer fills up, or if the current request is not contiguous to the previous one, the packetizer is sent to the file system for writing to the disk.

Figure 6 plots the performance of BUDDY, STREAM, and STREAM-PACKETIZER as a function of disk utilization. When disk utilization is high (around 95%), STREAM and STREAM-PACKETIZER perform comparable to BUDDY. This is because, at 95% utilization there do not exist long sequential portions of free space, and thus STREAM and STREAM-PACKETIZER can not perform long sequential write operations. On the contrary, when disk utilization is less than 72%, STREAM performs two times better than BUDDY, and STREAM-PACKETIZER performs 2.5 times better than BUDDY. Actually, STREAM-PACKETIZER manages to achieve more than 350 URL-get operations per second. To deliver their high performance, STREAM and STREAM-PACKETIZER need about 30% more disk space than the actual size of the URLs they need to store. Fortunately, the cost of disk space decreases rapidly (by a factor of two) every year [10]. Recent measurements suggest that most file systems are about half-full on the average [11], and

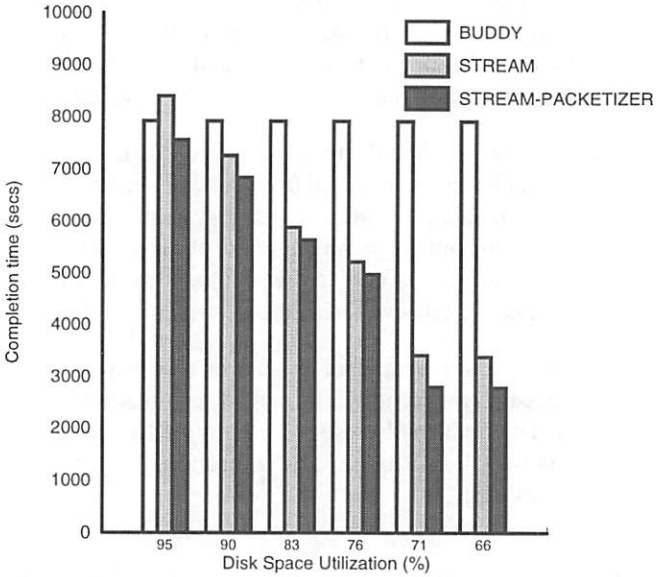


Figure 6: **Overhead of file management algorithms as a function of disk (space) utilization.** The figure displays the time it took to serve 1,000,000 file system operations as a function of disk utilization. The performance of STREAM and STREAM-PACKETIZER improves as disk utilization decreases. When disk utilization is around 70% both STREAM and STREAM-PACKETIZER outperform BUDDY by 2-3 times.

thus, log-structured approaches for file management may be more attractive than ever.

3.6 Improving Read Requests

Thanks to the STREAM and STREAM-PACKETIZER algorithms, URL-write operations suffer little (if any at all) seek and rotational overhead. However, URL-read operations still suffer from disk seek and rotational overhead, because the disk head must move from the point it was writing data to disk to the point it must read data. To make matters worse, once the read operation is completed, the head must move back to continue streaming its data onto the disk. Thus, each read operation (which necessarily happens within a stream of write operations) induces *two* head movements. To reduce this overhead we have developed a LAZY-READ approach which is much like STREAM-PACKETIZER with the following difference:

> Once a URL *read* operation is issued, it is being sent into an intermediate buffer. When the buffer fills up with read requests it forwards them to the file system, sorted according to the position (in

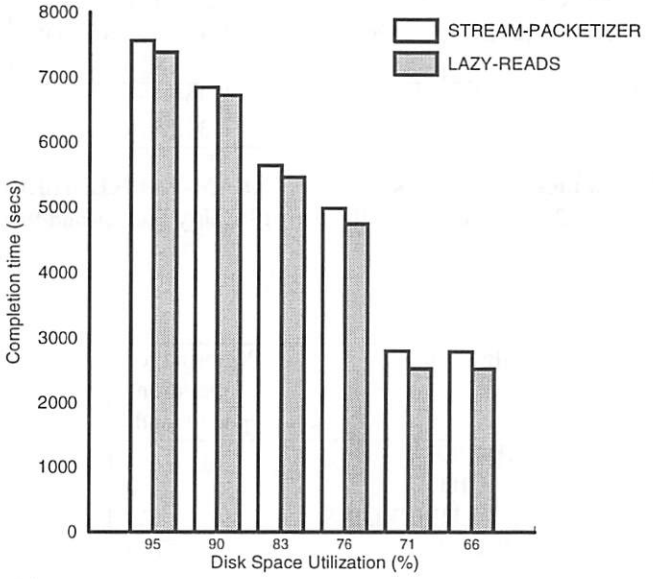


Figure 7: **Performance of LAZY-READS**. The figure displays the cost of serving 1,000,000 file system operations as a function of 2-Gbyte disk utilization. LAZY-READS gathers reads requests ten-at-a-time and issues them all at the same time to the disk reducing the disk head movements between the write stream and the data read. The figure shows that LAZY-READS improves the performance of STREAM-PACKETIZER by 10%.

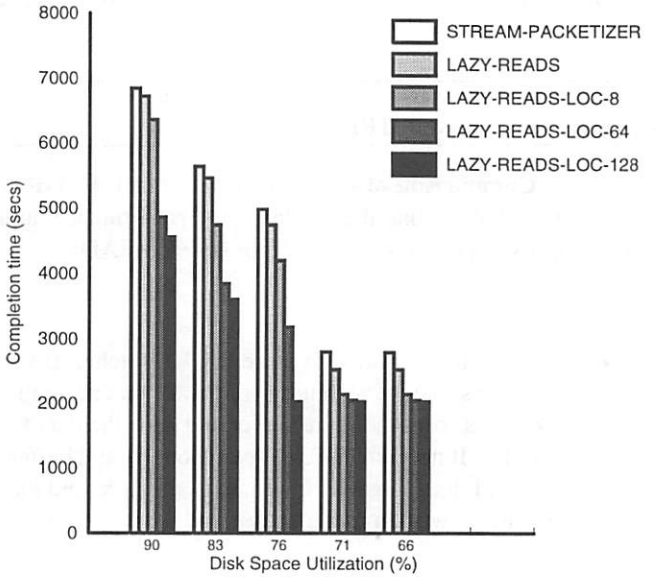


Figure 8: **Performance of LAZY-READS-LOC**. The figure displays the cost of serving 1,000,000 file system operations as a function of disk utilization. LAZY-READS-LOC attempts to put URLs from the same server in nearby disk locations by clustering them in locality buffers before sending them to the disk. We see that even as few as eight buffers improve performance over LAZY-READS.

the file) of the data they want to read.

Figure 7 shows that LAZY-READS improves the performance of STREAM-PACKETIZER by 10%. It is true that we expected a larger performance improvement. We traced the operating system actions and found that even if LAZY-READS sends read operations to the file system ten-at-a-time, the file system does not preserve this clustering and sent 3-5 clustered read operations to the disk in the average. Nevertheless, clustering read operations has potential and should be further explored. [6]

[6]The careful reader will notice however, that LAZY-READS may increase operation latency. Our trace measurements show that STREAM-PACKETIZER augmented with LAZY-READS is able to serve 10-20 read requests per second (in addition) to the write requests. Thus LAZY-READS will delay the average read operation only by a fraction of the second. Given that the average web server latency is several seconds long [2], LAZY-READS impose an unoticeable overhead. To make sure that no user ever waits an unbounded amount of time to read a URL from the disk even in an unloaded system, LAZY-READS can also be augmented with a time out period. If the time out elapses then all the outstanding read operations are sent to disk.

3.6.1 Preserving the Locality of the URL stream

The URL requests that arrive in a web proxy contain a significant amount of locality. For example, consider the case of an HTML page that has several embedded images. Every time a user requests that HTML page, (s)he will probably request all the embedded images as well. Thus, it may be worthwhile to store the HTML page and its embedded images in nearby disk locations so that future accesses to the HTML page and its embedded images will proceed at top speed. Unfortunately, current proxy servers tend to destroy such locality because they receive (and interleave) requests from several web clients. Thus, contiguous requests from a single web client may be received by the proxy server interleaved with tens of requests from other clients. Therefore, URLs that correspond to contiguous requests from a single client may be stored in the magnetic disk hundreds of Kbytes away from each other. To remedy this problem we have augmented the LAZY-READS policy with a number of locality buffers (LAZY-READS-LOC) that work as follows:

- There exist a set of locality buffers whose purpose is to accumulate URL-write operations that correspond to URLs from a single web server.

Algorithm	STREAM-PACK	STREAM-PACK +LAZY-READS	STREAM-PACK +LOC-128	STREAM-PACK +LAZY-READS +LOC-128
Time (minutes:sec)	46:31	42:08	36:16	33:51
Improvement (over STREAM-PACK)		10%	28%	37.5%

Table 1: **Comparison of the improvement of LAZY-READS, and locality buffers on the STREAM-PACKETIZER algorithm.** We see that the single largest performance improvement (28%) comes from the use of locality buffers and the next improvement (10%) comes from LAZY-READS.

- When the proxy wants to store a URL fetched from some web server, it searches for a buffer that accumulates URLs from the same server and adds the data to the buffer. If no such buffer is found, one victim buffer is selected, its contents are written to the disk, and the new URL is written in the buffer.

This policy gathers URLs from the same server into the same locality buffer, so that URLs from the same server requested within a short time interval will probably be written in contiguous file locations. We have evaluated the performance of this policy (for 8-128 locality buffers) against the performance of LAZY-READS and STREAM-PACKETIZER. Figure 8 plots the results. We see that the existence of even eight locality buffers (LAZY-READS-LOC-8) improves performance over LAZY-READS significantly. The most spectacular improvements happen at medium to large disk utilization. For example, at 76% disk utilization LAZY-READS-LOC-128 performs 2.5 times better than LAZY-READS. In all cases, however, LAZY-READS-LOC-128 is at least 30% better than LAZY-READS. In the best case LAZY-READS-LOC achieves around 500 URL-get operations per second.

In our final experiment we will explore what is the contribution of each factor (read-clustering/LAZY-READS and locality buffers/LOC) to the performance of STREAM-PACKETIZER. Table 1 presents the completion time of policies STREAM-PACKETIZER, STREAM-PACKETIZER augmented with LAZY-READS, STREAM-PACKETIZER augmented with locality buffers (128 of them), and finally, STREAM-PACKETIZER augmented with both LAZY-READS and locality buffers at 71% disk utilization. It also shows the (percentage) improvement of every method on top of STREAM-PACKETIZER. We see that LAZY-READS improve 10% on STREAM-PACKETIZER, locality buffers improve 28% on STREAM-PACKETIZER, and both methods improve 37% on STREAM-PACKETIZER.

Summarizing, table 2 shows the (best) performance of the various algorithms studied.

Algorithm	Performance (operations per second)
SIMPLE	14
SQUID	20
MULTIPLE-DIRS	23
BUDDY	133
STREAM	295
STREAM-PACK	358
LAZY-READS	396
LAZY-READS-LOC	495

Table 2: **Comparative performance (in terms of URL-get operations per second) of various file space management algorithms.**

4 Summary-Conclusions

In this paper we study the disk I/O overhead of world-wide web proxy servers. Using a combination of experimental evaluation and simulation based on traces from busy web proxies we show that web proxies experience significant overheads due to disk I/O. We propose several file management methods (like BUDDY, STREAM, LAZY-READS, STREAM-PACKETIZER, and locality buffers) which reduce the disk management overhead by more than a factor of 25 overall (from SQUID to LAZY-READS-LOC). Based on our experiments we conclude:

- The single largest source of overhead in traditional web proxies is the file creation and file deletion overhead associated with storing each URL on a separate file. Storing several URLs per file improves performance by an order of magnitude.
- Disk accesses of web proxies are dominated by write requests. Streaming these write operations to disk (much like log-structured file systems do) improves performance by a factor of 2-3.
- Web clients display a locality of reference in their ac-

cesses. Web proxies tend to destroy it by interleaving requests from several clients. Preserving this locality of reference results in better layout of URLs on the disk, which improves performance by 30%-150%.

- User-level file management policies improve performance (over traditional web proxies like SQUID) by a factor of 25 overall, leaving little space for improvement by specialized kernel-level implementations.

We believe that our results are significant today and they will be even more significant in the future. As disk bandwidth improves at a much higher rate than disk latency [10], methods that reduce disk head movements and stream data to disk will result in increasingly larger performance improvements.

Acknowledgments

This work was supported in part by the Institute of Computer Science of Foundation for Research and Technology -Hellas, in part by the University of Crete through project "File Systems for Web servers" (1200), and in part by EPET II project "E-Commerce" funded through the General Secretariat for Research and Development. We deeply appreciate this financial support.

Panos Tsirigotis was a source of inspiration and of many useful comments. Manolis Marazakis and George Dramitinos gave us useful comments in earlier versions of this paper. Katia Obraczka (our shepherd) provided useful comments in the final version of the paper. P. Cao provided one of the simulators used. We thank them all.

References

[1] M. Abrams, C.R.Standridge, G. Abdulla, S. Williams, and E.A. Fox. Caching Proxies: Limitations and Potentials. In *Proceedings of the Fourth International WWW Conference*, 1995.

[2] J. Almeida and P. Cao. Measuring Proxy Performance with the Wisconsin Proxy Benchmark. *Journal of Computer Networks and ISDN Systems*, 30:2179–2192, 1998.

[3] Azer Bestavros. Speculative Data Dissemination and Service to Reduce Server Load, Network Traffic and Service Time for Distributed Information Systems. In *Proceedings of ICDE'96: The 1996 International Conference on Data Engineering*, March 1996.

[4] Azer Bestavros. Using speculation to reduce server load and service time on the WWW. In *roceedings of CIKM'95: The Fourth ACM International Conference on Information and Knowledge Management*, November 1995.

[5] Trevor Blackwell, Jeffrey Harris, and Margo Seltzer. Heuristic Cleaning Algorithms for Log-Structured File Systems. In *Proceedings of the 1995 Usenix Technical Conference*, January 1995.

[6] A. Brown and M. Seltzer. Operating System Benchmarking in the Wake of Lmbench: A Case Study of the Performance of NetBSD on the Intel x86 Architecture. In *Proc. of the 1997 ACM SIGMETRICS Conference*, pages 214–224, 1997.

[7] Pei Cao and Sandy Irani. Cost-Aware WWW Proxy Caching Algorithms. In *Proc. of the USENIX Symposium on Internet Technologies and Systems*, 1997.

[8] Anawat Chankhunthod, Peter B. Danzig, Chuck Neerdaels, Michael F. Schwartz, and Kurt J. Worrell. A Hierarchical Internet Object Cache. In *Proc. of the 1996 Usenix Technical Conference*, 1996.

[9] G. Chisholm. Squid Performance as a Factor of the Number of Disks Utilised. http://www.dnepr.net/Squid/Benchmarking/Number-of-Disks/.

[10] M. Dahlin. *Serverless Network File Systems*. PhD thesis, UC Berkeley, December 1995.

[11] J.R. Douceur and W.J. Bolosky. A Large-Scale Study of File System Contents. In *Proc. of the 1999 ACM SIGMETRICS Conference*, pages 59–70, 1999.

[12] S.L. Fritchie. The Cyclic News Filesystem: Getting INN To Do More With Less. In *Proc. of the 1997 Systems Administration Conference*, pages 99–111, 1997.

[13] J. Gwertzman and M. Seltzer. The Case for Geographical Push Caching. In *Proceedings of the 1995 Workshop on Hot Operating Systems*, 1995.

[14] R. Hagmann. Reimplementing the Cedar File System using Logging and Group Commit. In *Proc. 11-th Symposium on Operating Systems Principles*, pages 155–172, 1987.

[15] J. Hartman and J. Ousterhout. The Zebra Striped Network File System. *Proc. 14-th Symposium on Operating Systems Principles*, pages 29–43, December 1993.

[16] D. Hitz, J. Lau, and M. Malcolm. File System Design for an NFS File Server Appliance. In *Proc. of the 1994 Winter Usenix Technical Conference*, pages 235–246, 1994.

[17] Cache Flow Inc. http://www.cacheflow.com.

[18] Inktomi Inc. The Sun/Inktomi Large Scale Benchmark. http://www.inktomi.com/inkbench.html.

[19] T.M. Kroeger, D.D.E. Long, and J.C. Mogul. Exploring the Bounds of Web Latency Reduction from Caching and Prefetching. In *Proc. of the USENIX Symposium on Internet Technologies and Systems*, pages 13–22, 1997.

[20] P. Lorenzetti, L. Rizzo, and L. Vicisano. Replacement Policies for a Proxy Cache, 1998. http://www.iet.unipi.it/~ luigi/research.html.

[21] C. Maltzahn, K. Richardson, and D. Grunwald. Performance Issues of Enterprise Level Web Proxies. In *Proc. of the 1997 ACM SIGMETRICS Conference*, pages 13–23, 1997.

[22] C. Maltzahn, K. Richardson, and D. Grunwald. Reducing the Disk I/O of Web Proxy Server Caches. In *Proc. of the 1999 Usenix Technical Conference*, 1999.

[23] E.P. Markatos. Main Memory Caching of Web Documents. *Computer Networks and ISDN Systems*, 28(7-11):893–906, 1996.

[24] E.P. Markatos and C. Chronaki. A Top-10 Approach to Prefetching on the Web. In *Proceedings of the INET 98 Conference*, 1998.

[25] M.K. McKusick and G.R. Ganger. Soft Updates: A Technique for Eliminating Most SynchronousWrites in the Fast FileSystem. In *Proc. of the 1999 Usenix Technical Conference - Freenix Track*, pages 1–17, 1999.

[26] L. McVoy and C. Staelin. lmbench: Portable Tools for Performance Analysis. In *Proc. of the 1996 Usenix Technical Conference*, pages 279–294, January 1996.

[27] Jeffrey C. Mogul. Speedier Squid: A Case Study of an Internet Server Performance Problem. *;login: The USENIX Association Magazine*, 24(1):50–58, 1999.

[28] M. Nelson, B. Welch, and J. Ousterhout. Caching in the Sprite Network File System. *ACM Transactions on Computer Systems*, 6(1):134–154, February 1988.

[29] V.N. Padmanabhan and J. Mogul. Using Predictive Prefetching to Improve World Wide Web Latency. *SIGCOMM Computer Communication Review*, 26:22–36, 1996.

[30] Athanasios Papathanasiou and Evangelos P. Markatos. Lightweight Transactions on Networks of Workstations. In *Proc. 18-th Int. Conf. on Distr. Comp. Syst.*, pages 544–553, 1998.

[31] J.E. Pitkow and M. Recker. A Simple, Yet Robust Caching Algorithm Based on Dynamic Access Patterns. In *Proceedings of the Second International WWW Conference*, 1994.

[32] Mendel Rosenblum and John Ousterhout. The Design and Implementation of a Log-Structured File System. In *Proc. 13-th Symposium on Operating Systems Principles*, pages 1–15, October 1991.

[33] A. Rousskov and V. Soloviev. On Performance of Caching Proxies. In *Proc. of the 1998 ACM SIGMETRICS Conference*, 1998.

[34] P. Scheuearmann, J. Shim, and R. Vingralek. A Case for Delay-Conscious Caching of Web Documents. In *6th International World Wide Web Conference*, 1997.

[35] M. Seltzer, M. K. McKusick, K. Bostic, and C. Staelin. An Implementation of a Log-Structured File System for UNIX. In *Proceedings of the 1995 Winter Usenix Technical Conference*, San Diego, CA, January 1993.

[36] V. Soloviev and A. Yahin. File Placement in a Web Cache Server. In *Proc. 10-th ACM Symposium on Parallel Algorithms and Architectures*, 1998.

[37] Joe Touch. Defining High Speed Protocols : Five Challenges and an Example That Survives the Challenges. *IEEE JSAC*, 13(5):828–835, June 1995.

[38] Stuart Wachsberg, Thomas Kunz, and Johnny Wong. Fast World-Wide Web Browsing Over Low-Bandwidth Links, 1996. http://ccnga.uwaterloo.ca/~ sbwachsb/paper.html.

[39] D. Wessels. Squid Internet Object Cache, 1996. http://squid.nlanr.net/Squid/.

[40] S. Williams, M. Abrams, C.R. Standbridge, G. Abdulla, and E.A. Fox. Removal Policies in Network Caches for World-Wide Web Documents. In *Proc. of the ACM SIGCOMM 96*, 1996.

[41] Roland P. Wooster and Marc Abrams. Proxy Caching that Estimates Page Load Delays. In *6th International World Wide Web Conference*, 1997.

[42] Michael Wu and Willy Zwaenepoel. eNVy: a Non-Volatile Main Memory Storage System. In *Proc. of the 6-th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 86–97, 1994.

Compression Proxy Server: Design and Implementation

Chi-Hung Chi, Jing Deng, Yan-Hong Lim
School of Computing
National University of Singapore
Singapore 119260
Email: chich@comp.nus.edu.sg

Abstract

Automatic data compression in the web proxy server is an important mechanism that can potentially reduce network bandwidth consumption and web access latency significantly. However, unlike traditional data compression, web protocols and data have unique characteristics that make compression challenging. These include data block streaming, wide range of data object sizes and types, and real-time response. In this paper, we focus on automatic web data compression in the HTTP proxy server. A new classification of web data compression based on system complexity and HTTP requirements is proposed: stream, block and file compression. Then, the concept of hybrid web data compression is introduced. To understand the potentials of web data compression better, an implementation of the proposed hybrid compression in the Squid proxy server is described. The result is very promising, as about 30% of the bandwidth can be saved easily. Furthermore, even with a low end Pentium 266 MHz PC as the proxy machine, the compression overhead is less than 1% of the transfer time.

1. Introduction

With the popularity of the Internet and the world wide web from academia to home and entertainment, network bandwidth has already become a scarce, valuable resource and an important obstacle to WWW surfing. Networks are getting more congested with an increasing amount of multimedia data, thus resulting in slower web response time and the comment of "World Wide Wait". To address this WWW bandwidth problem, one good approach is to compress web multimedia data on the network.

Compression of web multimedia data can be achieved in one of two ways: explicit and implicit. In the explicit case, it is the responsibility of the web data owner to store the compressed version of the data in the web server. Since HTTP 1.1 provides a compression data MIME type [12], any client browser that supports HTTP 1.1 can view the compressed data automatically. For network management, this is the simplest approach, because compression is completely transparent to it. However, this causes inconvenience to content management because any maintenance or update of web information must perform explicit data decompression. More importantly, unless the compression format is a standard type supported by HTTP, implicit collaboration between the web client and server for compression might have a certain degree of difficulty. The existence of a tremendous amount of uncompressed web data also raises the demand for automatic web data compression. Finally, non-standard and older versions of browsers might not support automatic decompression.

Under the implicit model, it is the web server and proxy server that will handle web data compression automatically. The price for this shifting, however, is a new technology and system architecture design that can overcome the inherited problems of web data compression such as data block streaming, the wide range of data object sizes and types, and real-time response. Just like the situation in the operating system and the hard-disk storage, automatic web data compression is possible, but is not as trivial as it appears.

In this paper, we will investigate the system architecture design and support for automatic web data compression in the HTTP proxy server. Due to the unique environment of the web, new requirements and constraints for web data compression will be identified first. In particular, the implications of data block streaming of web information will be investigated. Then, hybrid data compression will be proposed. This hybrid compression will be implemented in the Squid proxy server architecture for the evaluation of bandwidth reduction and the overhead incurred. The result is very promising, as about 30% of the bandwidth can be saved easily. Furthermore, even with a low end Pentium 266 MHz PC as the proxy machine, the compression overhead is less than 1% of the transfer time. This is important because compression is completely transparent to the web surfers and can work co-operatively with other bandwidth saving mechanisms.

2. Constraints & Implications of Network Compression

In the WWW environment, there are unique characteristics of web information and constraints of the proxy and web servers that make automatic networked multimedia data compression challenging. Some of the major ones are listed below and their implications on web data compression are discussed.

Data Object Sizes

The size distribution of web objects ranges from a few hundred bytes to tens/hundreds of thousand bytes. More importantly, due to the practice of web page design, there is a significant portion of the web objects whose sizes are quite small. This makes web data compression challenging. It is because traditional data compression algorithms are much less effective towards small sized files. It is also found that the occurrence frequency of repeated character strings in a web object (such as an HTML file) might not be very high (because of the limited size). One the other hand, this frequency turns out to be very high across multiple web objects. A good example is HTML script marks. The HTML language syntax has a very limited set of character strings and they appear in almost every web page. Hence, it will be beneficial to have a static table that will map the frequently used HTML strings into variable size tokens.

Data Object Types

Another unique characteristic of web objects is the wide range of data types. There are simple HTML text files, application/octet-streams, gif/jpeg images, avi/asf/mpeg videos, JAVA applets, executables, etc.. With different data encoding formats and information entropy, the compressibility of these object types varies significantly. Furthermore, some object types might allow lossy compression (e.g. images) while the others insist on lossless compression (e.g. HTML files). This feature makes a single universal data compressor difficult to find (if at all possible). As a result, hybrid data compression, with both lossy and lossless compressors, is likely be the direction to handle such a wide variety of object types.

Data Block Streaming

This is one of the toughest constraints that the web imposes on networked multimedia data compression. According to the current HTTP protocols, data is streamed to the client in blocks. This is to improve the client's perception of the web object retrieval time and to reduce system resource consumption. Data block streaming implies that the compression/decompression process can only work on those blocks that are in the buffer space of the HTTP proxy (not necessarily the entire object). As a result, for those data compressors that require multiple passes on the object content, extra effort will be required to save the data being streamed in the server before the (de-)compression process can take place. Although single-pass data compressors are available, their compression effectiveness is not as good as those of multi-pass ones.

3. Classification of Web Data Compression

In this section, we propose a new classification for web data compression that is defined in terms of the system resource requirements and data block streaming in the current HTTP protocol. There are basically three classes of compression approaches for the web:

[1] *Whole File Compression*
Under this class, the compression needs to work on the entire file of a web object. There is no compression/decompression on the partial object data.

[2] *Data Block Compression*
This class of compression processes the information that is stored in the HTTP proxy buffer. Since a web object is made up of one or more data blocks, compression on a web object implies individual compression and decompression processes on data blocks that appear in the proxy buffer.

[3] *Data Stream Compression*
As its name implies, this class of compression treats data as a continuing stream. Whenever a proxy server receives some data, it will compress/decompress the data immediately and the result will then be passed to the next level of the network. There is no need to buffer data for future compression/decompression.

Based on the different stages of triggering of the compression process, each of these three approaches have different implications on the complexity of the proxy system and the compressor. This is given in Table 1. From this table, we see that data block compression is more suitable for web multimedia data in the HTTP proxy than either the stream or whole file compression approaches. Under this approach,

additional data buffering is minimal, as the proxy also needs to keep the data of the currently transferred blocks in the buffer anyway. The maximum data size stored in the buffering space implies that data does not need to be stored on disk; thus no additional disk operation is expected. The extra access delay time, as perceived by the web surfer, is also not important because this delay time is defined by the fixed buffer size.

	Stream	Block	Whole File
Additional Memory Buffering Req.	No	No	Resource to hold entire web object
Additional Disk Operations	No	Not expected	Yes
Partial Web Data Transfer	Same	In size of proxy buffer	No
Implementation Complexity	Slightly increased	Most complicated	Increased
Compression Efficiency	Lowest	Close of that of whole file compression	Highest
Choice of Compression Algorithms	Only single pass	All kinds	All kinds

Table 1: Comparison of Three Web Compression Types (With Respect to the Current Proxy Requirements)

As far as compression algorithms are concerned, data block streaming can incorporate all kinds of compressors, including multi-pass ones. The tradeoff, however, is the complexity of the proxy server architecture. Among these three classes, data block compression interferes with the proxy data flow the most, resulting in the most complicated system structure. There will also be limitations to the applicability of multi-pass compressors for web data: a multi-pass algorithm can only work on the object that can fit in the proxy buffer. If the object size is greater than the buffer size, either one of these two situations will happen. For web data objects whose blocks can be compressed independently (e.g. `text` files), performance will be lost slightly. It is because a large file is now divided into smaller segments, each of which is compressed independently. However, for those objects whose blocks cannot be compressed independently (e.g. `jpeg`), data block compression will not work.

4. Algorithm Selection

In Section 2, we mentioned that it is important to have hybrid compressors, each of which is optimized for one predefined type of data object. In this section, we would like to propose a sample selection of data compressors for the three compression approaches mentioned above. This will help in the testing and evaluation of the compression proxy potentials. *Note that this is just a reasonable set of combinations and may not be optimal. There is no intention in this paper to define any optimal algorithms.*

Let us look at each object type and see what kind of compression can be performed under the three approaches of web data compression:

`gif` *Objects*

It is important to compress `gif` objects because about one third of the web bandwidth consumption is due to `gif`. To compress these objects, we propose to use the GIF-to-JPEG transformer with 25% lossy factor. Lossy compression is used here because most `gif` objects are for decoration purposes and can tolerate some degree of loss. Furthermore, the loss in the image quality due to compression is very small to be noticed by web surfers. Due to the multi-pass nature of the `gif-to-jpeg` transformer, it cannot be used in data stream compression. Furthermore, for data block compression, there are two additional requirements:

- If the `gif` object size is greater than the proxy buffer size, no `gif-to-jpeg` compression will be performed. This is because the transformer cannot work on partial images.
- If the `gif` object size is less than 4 KBytes, no `gif-to-jpeg` compression will be performed. Table 2 shows the compression ratio of the `gif-to-jpeg` transformer with 25% lossy factor. From this table, we see that the `gif-to-jpeg` transformer actually expands a `gif` file if the file size is between 0 and 4 KBytes.

`gif` Object Size Range	Compression Ratio
0 – 1 K	0.3322
1 – 4 K	0.7891
4 – 8 K	1.3136
8 – 16 K	1.9825
16 – 32 K	2.9885
32 – 64 K	5.6460
64 – 128 K	9.9781
> 128 k	18.6050

Table 2: Compression Ratio of `gif-to-jpeg`

`text/octet-stream` *Objects*

The choice of compressors for these objects is relatively simple because many text compressors can

work very well on them. The only complication is the complexity of the compression algorithm: whether multi-pass algorithms should be allowed. For data block compression, we use the ZLIB library [5]; for whole file compression, we suggest using GZIP. They are some of the most common compressors for text. For stream compression, LZW is used instead because it is a single pass compressor. Furthermore, for HTML objects, mapping of the HTML script marks to variable size tokens is performed with the help of a predefined table.

`jpeg` *Objects*

Unlike `gif` objects, no transcoding compression is done on `jpeg` objects for all the three types of compression. There are a few reasons for this. Firstly, the `jpeg` objects are already in compressed format. It will be relatively difficult to compress the objects further without losing image quality substantially. Secondly, it is observed that on the web, `jpeg` objects are used much less often for decoration than `gif` objects are; hence losing image quality for further `jpeg` transcoding might result in user dissatisfaction. Thirdly, the algorithm to change from normal `jpeg` to progressive `jpeg` is a multi-pass algorithm. Since the size of a `jpeg` object is usually larger than the size of the HTTP proxy buffer, it will not have a significant effect on data block compression. Note that just like the `gif-to-jpeg` compression, `jpeg` transcoding is possible as long as the user is willing to trade off the image quality for bandwidth.

Others

For objects other than those mentioned above, no compression will be performed. It is because the percentages of their bandwidth consumption are not high and their optimal compressors are unknown.

5. Proxy-To-Proxy Compression in Squid

To get a better understanding of the design feasibility of automatic web data compression in the proxy server, we are going to describe an implementation of the block compression mechanism between two Squid proxy servers in this section. In this implementation exercise, the Squid proxy version 2.1 [23] is chosen to be the basic platform for experimentation. In Squid, data is transferred block by block. Whenever a Squid proxy receives data from its upper web server or proxy level, it will send the data to the client/proxy at the next network level as soon as possible. To make our discussion easier, the following terms are used. The "*compression proxy*" is the proxy server that does the compression of web data and then sends it to the decompression proxy server. The "*decompression proxy*" is the proxy server that receives the compressed data from the compression proxy and decompresses it before it sends the data to its client.

5.1. Implementation Considerations

During the implementation of data block compression in the Squid proxy server, there are at least three design issues that need to be handled properly.

5.1.1. Encoding of Compression Messages

In the implementation of proxy data compression, it is extremely important to handle the handshaking mechanism between two proxy servers properly. On one end, the decompression proxy needs to insert a message into the request header to notify the compression proxy that it has the decompression capability. The compression proxy will also need to write a message in the reply header to indicate that the entity body is compressed. Furthermore, both proxies need to delete these additional messages in the request and reply headers before they pass the request to the next level of the network. According to the HTTP/1.1 protocol [12], we propose the following solution:

- The decompression proxy adds a "`Transfer-Encoding: Block-Decoding`" field in the request header to notify the compression proxy what compression method(s) it supports.

- The compression proxy sends a "`Transfer-Encoding: Block-Encoding`" field in the reply header to notify the decompression proxy what compression method it uses to compress the data.

One important set of parameters that needs to be communicated between the two proxy servers is the compressed block size and the uncompressed block size. In the block compression on Squid, data will be compressed in blocks; thus the decompression proxy server will not receive the same number of bytes as in the original uncompressed block. Even worse, it is possible that the decompression proxy buffer might receive more than one block of data at a time, or a whole compressed data set is received in multiple blocks. To solve this problem, we propose to add a

four bytes "block header" to each data set of compressed blocks in the proxy buffer. The first two bytes of the header record the size of the compressed set while the last two bytes record the original (uncompressed) data set size.

With this information, the decompression proxy will use the first two bytes to separate each compressed data set from the whole trunk of the received data. It will also use the last two bytes of the header for decompression. Furthermore, if the last two bytes record "0", it will mean that the data block is not compressed. This happens either when the compression proxy cannot perform compression on that data type or it is not beneficial to perform compression on the given data (as we mentioned previously in Section 4). With the two bytes mechanism to record the data sizes, the maximum working set size is bounded by 64 KBytes. As will be shown later, the recommended buffer size for data block compression is 32 Kbytes. Hence, the two bytes mechanism should be sufficient in most practical implementations. In the case where a really large block size is used, more bytes can be allocated to hold this information.

Another problem in the hand-shaking mechanism is the length of the transfer file. The web server records the length of the transfer file in the reply header field "Content-Length". Once the proxy performs compression or decompression, the actual length of the file will definitely be affected. To handle this situation, our decompression proxy will not check the incoming file length with the "Content-Length" field. That is, even though the compression proxy server transfers an object with file length less than the value of the "Content-Length" field, no error will be reported.

5.1.2. Memory Allocation

To call the "compress()" and "uncompress()" functions in the proxy server, some working memory will be required. Furthermore, the decompression proxy also needs memory to keep the compressed data block. All these can be handled by the group of memory management functions provided by Squid; we use the xmalloc() and xfree() functions to allocate and free memory in the compression process.

5.1.3. Data Structure

To complete the support for automatic web proxy compression, the compression status information of a web request needs to be reflected in its data structure inside the proxy server. In Squid, the request data and the reply data of a request are linked together through the data structure "_HttpStateData". It records all the necessary information about the request and the reply. This structure is generated when the proxy processes a request and it will be used during the entire request and reply processes. In the implementation of our compression proxy, we added a field "need_compress" in the "_HttpStateData" structure to specify that this reply can be compressed. We also added another field "can_compress" in the _StoreEntry structure to indicate that if reply data type is actually in the compressed format. On the decompression proxy side, the "_StoreEntry" structure records the information of the reply. It is linked to the structure "_HttpStateData". Another variable, "can_decompress" is also added as the compression flag in this structure. If the decompression proxy finds that the incoming data is in compressed format, the flag will be set.

5.2. Workflow of Squid Compression Proxy

In this section, we would like to put together the actual workflow path of a web request in the Squid compression proxy environment. This not only helps us to understand the complete design of the Squid compression proxy server, but it also allows us to appreciate the design decisions and to see how various techniques fit together in a single system. The modified workflow for Squid compression proxy is summarized below.

Workflow in the Squid Decompression Proxy

Step A1:
When the Squid decompression proxy receives a request from a client, it will add a "Transfer-Encoding: Block-Decoding" field in the request header.
(Afterwards, it will wait for data to come back from the compression proxy).

Step A2:
When the decompression proxy receives the reply data from the compression proxy, it will analyze the reply

header. If it finds that a "Transfer-Encoding: Block-Encoding" field is set and the "Content-Type" matches its decompressor, it will set the variable "StoreEntry->can_decompress".

Step A3:
When the "StoreEntry->can_decompress" field is set, the decompression proxy will read the compressed and the uncompressed set sizes from the "block header". If the incoming data set size is greater than or equal to the compressed block size, the decompression proxy will extract one compressed block, then call the function "uncompress()", and afterwards go back to step *A3*; otherwise the data will be saved in StoreEntry->keep_buf.

Step A4:
If the can_decompress field is set, the decompression proxy will erase the "Transfer-Encoding: Block-Encoding" field from the reply header.

Workflow in the Squid Compression Proxy

Step B1:
If the request header has a "Transfer-Encoding: Block-Decoding" field, the compression proxy will set the variable "_HttpStateData->need_compress". Then, it erases the "Transfer-Encoding: Block-Decoding" field from the request header.

Step B2:
The compression proxy checks the reply header field "Content-Type". If this field matches with the proxy's supported compression data type(s), both the "HttpStateData->need_compress" and "StoreEntry->can_compress" variables will be set.

Step B3:
When the "StoreEntry->can_compress" field is set, the compression proxy will call the function "compress()". It also writes the set size of the compressed and the uncompressed blocks in the header of the data block.

Step B4:
It adds the "Transfer-Encoding: Block-Encoding" field in the reply header and will then send the compressed data block to the decompression proxy.

6. Experimental Results

To measure the effectiveness of web multimedia data compression in the HTTP proxy, we repeated the web surfing pattern of the proxy trace that we collected in a junior college in Singapore. The proxy server used was the Squid proxy on a low-end DEC Alpha workstation. The proxy trace log was collected for about one year, and the standard proxy information was recorded. The workload of the proxy was about 5,000 to 20,000 requests per day and the school had a leased line of 128Kbps

Two sets of experiment were conducted with the same sequence of web object references:

- *client browser ⇔ original SQUID proxy ⇔ original SQUID proxy ⇔Web server*
- *client browser ⇔decompression SQUID proxy ⇔ compression SQUID proxy ⇔Web server*

Each of the four proxy servers (either with or without compression) was Squid version 2.1 with 30 Mbytes of allocated memory and 200 Mbytes of cache space. The proxy server was run on a dual Pentium II 266 MHz PC with 64 MBytes memory and 128 MBytes swap space and the operating system was LINUX. For each set of experiments, both proxy servers were run on the same machine. We did this because the overhead of compression and decompression could be estimated better by avoiding any data transfer between the two testing proxy servers in the public network. In our experiments, we used one day's web access sequence, which consists of about 10,000 requests. The access latency of the web object and the bandwidth consumption were measured. The overhead of compression/decompression in the proxy can be estimated by comparing the time consumed by the two sets of experiments.

Before we discuss the performance of the three different compression approaches, it will be helpful to have statistical data on the web objects that pass through the proxy server.

6.1. Distribution of Web Object Types

Web objects have a wide variety of data types. A data type distribution of WWW objects provides hints on what kind of compression algorithms should be supported and how the proxy compression system should be structured and optimized. We collected this statistic by extracting the "file type" and "file

size" fields of entries in the proxy trace. The result is shown in Figure 1. The percentage is defined in terms of the bytes transferred instead of the number of objects requested.

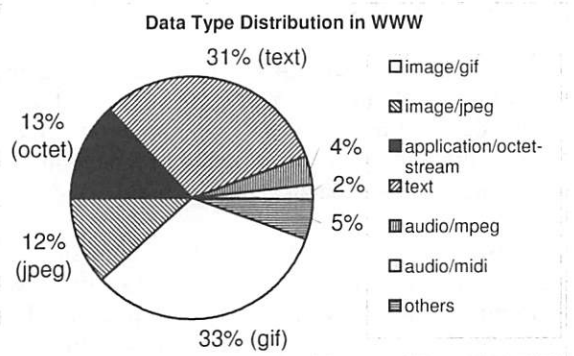


Figure 1: Distribution of Web Object Types (in terms of the Bytes Transferred)

From the figure, we see that the main data types of web objects are: `gif` data (33%), all kinds of `text` data (31%), `octet-stream` data (13%), and `jpeg` image data (12%).

6.2. Distribution of Web Object Sizes

Understanding the distribution of web object sizes is important to estimate the overall benefits of web data compression. Given a web object type, the size distribution can indicate how effective the data compression will be. Usually, small objects are much more difficult to compress than large ones. Due to data block streaming in the web environment, the statistic also shows how often a web object can fit into the HTTP proxy buffer.

File Size in Bytes	Gif	text	octet
0 – 1 K	3.77%	1.32%	0.04%
1 – 4 K	15.25%	7.75%	0.66%
4 – 8 K	20.92%	16.96%	0.69%
8 – 16 K	28.89%	30.15%	1.14%
16 – 32 K	14.12%	28.75%	0.89%
32 – 64 K	8.44%	9.30%	1.53%
64 – 128 K	5.00%	2.79%	2.40%
> 128 K	3.60%	2.99%	92.66%

Table 3: File Size Distribution of `gif`, `text`, and `octet-stream` (in terms of Bytes Transferred)

Table 3 shows the distribution of the amount of data retrieved under an object size range for a given object type (again, not the number of files). It shows that for `text` objects, most of the data are clustered in the range of 4 KBytes to 32 KBytes. This is a good range for most text compressors to have reasonable performance. The size distribution of the `gif` objects is quite similar to that of `text` objects except that there is a higher percentage of data with file sizes under 4 KBytes. As was discussed in the previous section, no aggressive transformation or compression will be done on this group of objects - the objects will be expanded upon compression. Data distribution of the `octet-stream` objects is quite different from the rest. Most of the objects are larger than 128 KBytes.

6.3. Compression Effectiveness

The results of compression effectiveness of the proposed approaches are given in Figures 2-4. The x-axis is the compression types: stream, block with various buffer sizes, and file. The results are extremely encouraging and illustrate the potentials of web data compression.

Figures 2(a)-(d) show the bandwidth saved by the three web data compression approaches. Overall, the saving from whole file compression is still the highest, with an average of about 37%. This is expected because the whole file is available for compression. With a reasonable HTTP proxy buffer size, the performance of data block compression is actually very impressive. For a block size of 32 KBytes, the bandwidth reduction is 32%; and with a 64 KBytes block size, the reduction is 34.39%. These are only a few percent lower than the upper limit. Moreover, the incremental performance gain in the bandwidth reduction starts to level off when the block size is greater than 32 KBytes. For data stream compression, about 14.66% of bandwidth reduction can be expected. This is quite good; however, it is only about half of what can be obtained from data block compression with 32 KBytes block size. This justifies our argument that data block compression is worth the increase in system design complexity.

Comparing Figure 2(a), 2(b) and 2(c), we see that `text` objects are the most compressible ones. Usually, 50% to 70% of the bandwidth consumption due to `text` objects can be saved. Also, it is interesting to notice that there is practically no difference in bandwidth saving for buffer sizes larger than 4KBytes. This further supports our choice for data block compression. For `gif` objects, the situation is completely different. With buffer size less than 4 KBytes, there is no bandwidth saving. This is expected because no compression takes place; compression only

increases the object size. With buffer size greater than 4 KBytes, the bandwidth reduction increases quite rapidly with the buffer size and then levels off after 64 KBytes. Note that although the compression ratio of `gif` to `jpeg` is very high, the bandwidth reduction is still bounded by about 43%. This is because there exists a set of `gif` objects (with size less than 4 KBytes) that will not benefit by the `gif-to-jpeg` transformation. The contribution of the `octet-stream` compression to the bandwidth reduction is the least. This is probably due to the lesser compressibility of the `octet-stream` objects.

Figure 3 shows the average time required to perform compression and decompression in the Squid proxy server. It shows that the compression overhead is actually quite small, with an average of about 5 milli-seconds per KBytes. Note that there is a drop in compression/decompression time for octet objects from data stream compression to 1 Kbytes data block compression. Careful investigation shows that this is due to the change of compressors from LZW to ZLIB. Furthermore, by the nature of the ZLIB algorithm, its decompression time is smaller than its compression time. Similar situation happens to text objects from 64 KBytes data block compression to whole file compression. Despite the similar cascading concept used by ZLIB and GZIP, the implementation of GZIP is more efficient than that of ZLIB.

Figure 4 shows the ratio of the compression time overhead to the original access latency (before compression). The overhead is indeed very small; the average is only about 0.6% of the access latency. With the consideration of a 30% reduction in the network bandwidth, the overall web access latency will definitely improve. Furthermore, the proxy machine used in the experiment is an outdated one. With a reasonable machine configuration for the proxy server, the compression overhead will be reduced further.

Finally, we estimated the improvement of web access latency due to data compression. The testing was done on a network segment that covered two local internet service providers and their internet exchange gateway. The result is shown in Figure 5. Stream data compression improves the web access latency by about 10%, which is quite good. With block data compression, the improvement goes up to about 25%-30%. This is expected because more data can be compressed and more effective two-pass compressors can now be used. However, for whole file compression, the improvement drops back to about 18%. This is mainly due to the buffering of the whole object for compression before it is sent to the decompression proxy server.

7. Related Work

Data compression [14,17,21] is a fundamental area of research in information theory. Traditional system design (including the operating system) and disk controllers provide a good foundation for supporting automatic data compression [3,6,7].

Despite the long history of data compression, the application of data compression to the web environment is relatively new. Nielsen et al. [18] investigated deflate data compression [8] from the viewpoint of a web server. Both Mogul et al. [16] and Velasco et al. [24] realized the potential benefits of compression in the HTTP proxy by compressing the web objects with text compressors such as GZIP. Santos et al. [22] looked at mechanisms to suppress the transfer of replicated data in the web environment. While their studies gave a good foundation for web data compression, there was no discussion on the constraints of the web environment, the design considerations of the compression proxy, and the implementation issues.

In the wireless mobile environment, there are research efforts to reduce the network bandwidth requirements for mobile devices and PDAs [25]. One representative project is the GloMop from Berkeley [9,10]. It achieved better end-to-end performance and higher quality display output for low-end clients through dynamic distillation. However, the data streaming nature of the WWW data cannot be handled by their techniques. Another approach to address the WWW bandwidth problem is to use delta encoding for the web response. When a cache object is outdated in the local proxy, only the delta of the change in content will be transferred from the server to the proxy. This idea was proposed in [2,11,13,26] and [16] later verified its potentials with realistic traces. The idea is good, but it only works on objects that are outdated in cache. This limitation is important to network bandwidth reduction because over half of web objects are referenced only once [15]. In network caching, researchers address the network bandwidth problem by keeping those web objects that are expected to be reused in the local memory or cache [1,4]. However, the network proxy cache cannot reduce the network bandwidth consumption for first time object accesses. To address this issue, the idea of prefetching [19,20] is proposed. The main difficulty in web prefetching is to have a very high prefetch accuracy; the spatial

property of web objects is not well defined and many web objects are referenced only once in cache [15].

8. Conclusion

In this paper, we investigated web data compression as the mechanism to solve the Internet bandwidth problem. Based on the unique features of the web protocol and data, we proposed a new classification for web data compression: stream, block, and file. We argued that data block compression fits into the web environment best because it handles the streaming of web data properly and allows multi-pass compressors to work on web data without explicit file storage. Then we proposed the hybrid web compression mechanism and implemented it on the Squid proxy server to test out its feasibility and to evaluate its performance. The result is very impressive; about 30% of the network bandwidth can be saved and the compression and decompression overhead is less than 1% of the web access latency (even on an outdated PC proxy server). The result is important because the compression process is completely transparent to the web surfers and it can work co-operatively with other bandwidth saving mechanisms.

Acknowledgements

The authors would like to thank P. Krishnan and Jessica Kornblum who helped to improve the final version of this paper. We would also like to thank the anonymous referees for their valuable comments on earlier drafts of this paper.

Reference

[1] Abrams, M., Standridge, C.R., Abdulla, G., Williams, S., "Caching Proxies: Limitations and Potentials," *Proceedings of the Fourth International World Wide Web Conference*, Boston, U.S.A., December 1995.

[2] Banga, G., Douglis, F., Rabinovich, M., "Optimistic Deltas for WWW Latency Reduction," *Proceedings of the 1997 USENIX Annual Technical Conference*, Anaheim, California, USA , January 1997.

[3] Burrows, M., Jerian, C., Lampson, B., Mann, T., "On-line Data Compression in a Log-Structured File System," *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1992, pp. 2-9.

[4] Chankhunthod, A., Danzig, P., Neerdaels, C., Schwartz, M.F., Worrell, K.J., "A Hierarchical Internet Object Cache," *Proceedings of the 1996 USENIX Annual Technical Conference*, January 1996.

[5] Deutsch, P., Gailly, J., "ZLIB Compressed Data Format Specification Version 3.3", RFC 1950, May 1996.

[6] Douglis, F., "On the Role of Compression in Distributed Systems," *Proceedings of the Fifth ACM SIGOPS European Workshop*, Mont St.-Michel, France, September 1992.

[7] Douglis, F., "The Compression Cache: Using On-line Compression to Extend Physical Memory," *Proceedings of 1993 Winter USENIX Conference*, San Diego, CA, January 1993, pp. 519-529.

[8] Deutsch, P., "DEFLATE Compression Data Format Specification version 1.3," RFC 1951, Aladdin Enterprises, May 1996.

[9] Fox, A., Brewer, E.A., "Reducing WWW Latency and Bandwidth Requirements by Real-Time Distillation," *Proceedings of the Fifth International WWW Conference*, May 1996.

[10] Fox, A., Gribble, S., Brewer, E.A, "Adapting to Network and Client Variation via On-Demand Dynamic Transcoding," *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996, pp. 160-170.

[11] Housel, B.C., Lindquist, D.B., "WebExpress: A System for Optimizing Web Browsing in a Wireless Environment," *Proceedings of the ACM Second Annual International Conference on Mobile Computing and Networking*, Rye, NY., November 1996, pp. 108-116.

[12] HTTP 1.1 Protocol. Request for Comments: 2616, June 1999.

[13] Hunt, J., Vo, K.P., Tichy, W., "An Empirical Study of Delta Algorithms," *Proceedings of the IEEE Software Configurations and Maintenance Workshop*, Berlin, March 1996.

[14] Lelewer, D., Jacobson, V., "Data Compression," *ACM Computing Surveys*, 19(3), 1987, pp. 261-296.

[15] Lim, S.N., "Live Range Modelling of WWW References," *M.Sc. Thesis, School of Computing, National University of Singapore*, 1999.

[16] Mogul, J.C., Douglis, F., Feldmann, A., Krishnamurthy, B., "Potential benefits of delta-encoding and data compression for HTTP," *Proceedings of the ACM SIGCOMM '97 Conference*, September 1997.

[17] Nelson, M., Gailly, J.L., *The Data Compression Book*, M&T Books, 1996.

[18] Nielsen, H.F., Gettys, J., Baird-Smith, A., Prud'hommeaux, E., Lie, H.W., Lilley, C., "Network Performance Effects of HTTP/1.1, CSS1, and PNG," 1997. http://www.w3.org/Protocols/HTTP/Performance/pipeline.html.

[19] Padmanabhan, V.N., Mogul, J.C., "Improving HTTP Latency," *Computer Networks and ISDN Systems*, 28(1-2), December 1995, pp. 25-35.

[20] Padmanabhan, V.N., Mogul, J.C., "Using Predictive Prefetching to Improving World Wide Web Latency," *Computer Communication Reviews*, 26(3), 1996, pp. 22-36.

[21] Salomon, D., *Data Compression: the Complete Reference*, Springer Press, 1997.

[22] Santos, J., Wetherall, D., "Increasing Effective Link Bandwidth by Suppressing Replicated Data," *Proceedings of the USENIX 1998 Annual Technical Conference*, June 1998.

[23] Squid Web Site. http://Squid.nlanr.net

[24] Velasco, J.R., Velasco, L.A., "Benefits of Compression in HTTP Applied to Caching Architectures," *Proceedings of the Third International WWW Caching Workshop*, 1998. http://wwwcache.ja.net/events/workshop/32/manchester.html.

[25] Wachsberg, S., Kunz, T.,. Wong, J., "Fast World Wide Web Browsing over Low Bandwidth Links," June 1996. Available as http://ccnga.uwaterloo.ca/~sbwachsb/paper.html.

[26] William, S., Abrams, M., Standridge, C.R., Abdulla, G., Fox, E.A., "Removal Policies in Network Caches for World-Wide-Web Documents," *Proceedings of the ACM SIGCOMM '96 Conference*, August 1996.

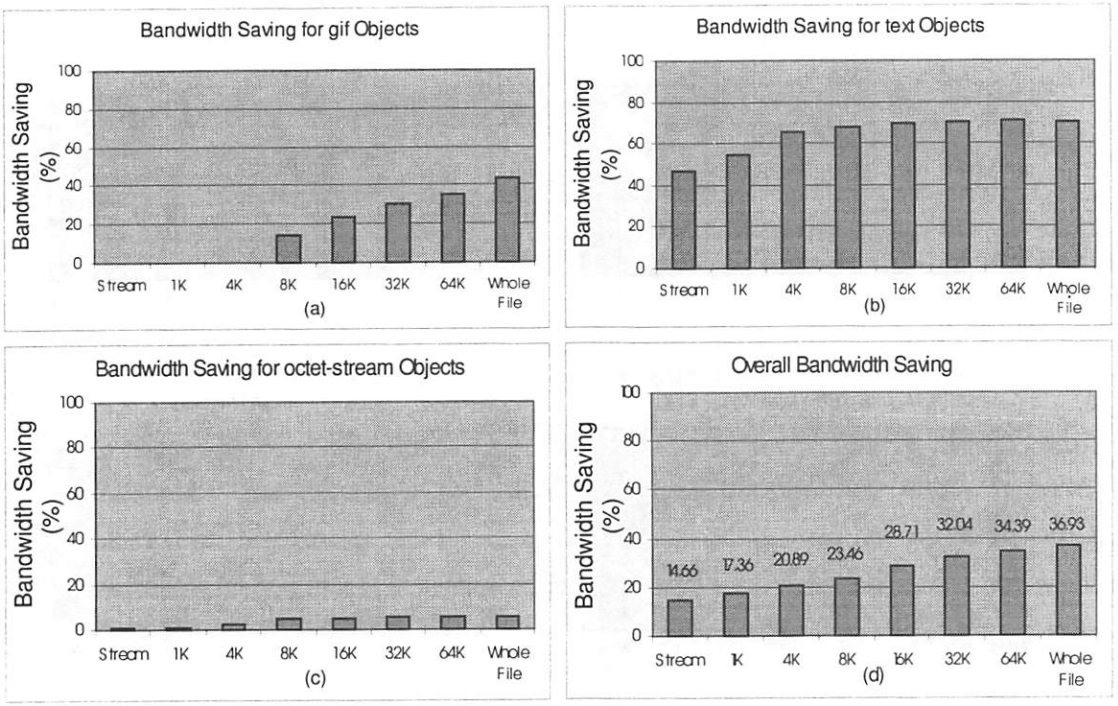


Figure 2: Bandwidth Saving By Web Compression
(x-axis: stream, block with various buffer sizes, file)

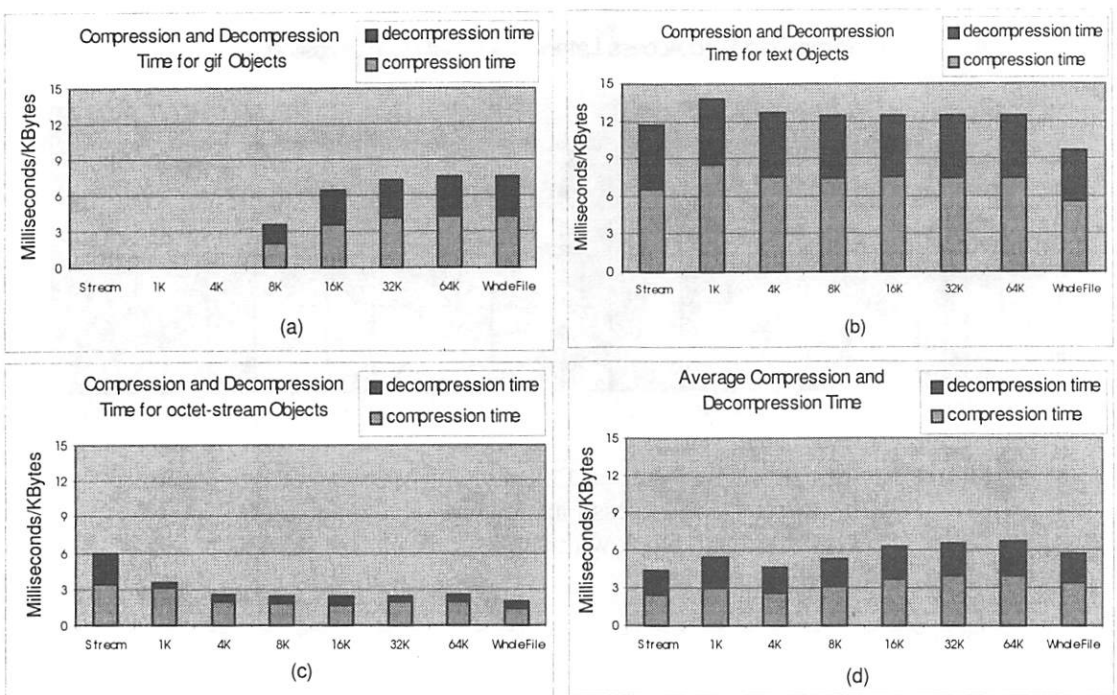


Figure 3: Compression and Decompression Overhead
(x-axis: stream, block with various buffer sizes, file)

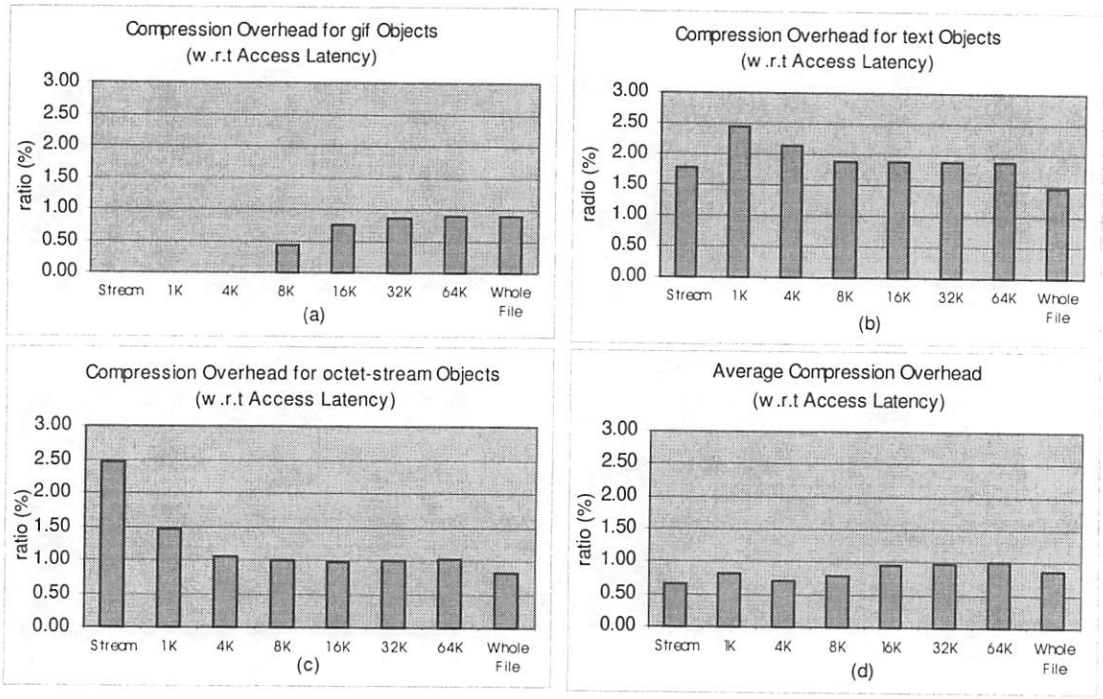


Figure 4: Normalized Web Compression Overhead
(with respect to the Access Latency)
(x-axis: stream, block with various buffer sizes, file)

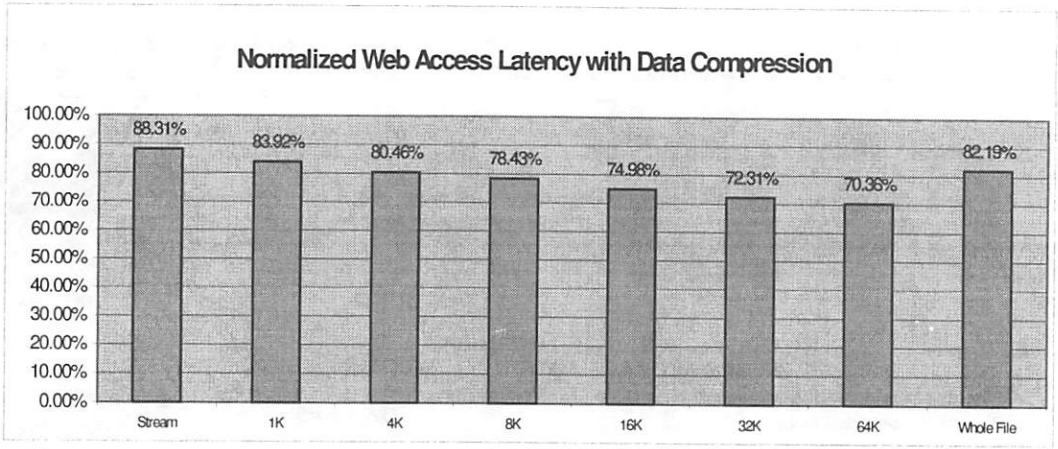


Figure 5: Normalized Web Access Latency with Web Data Compression
(with respect to the Access Latency without Compression)
(x-axis: stream, block with various buffer sizes, file)

On the Performance of TCP Splicing for URL-aware Redirection

Ariel Cohen Sampath Rangarajan Hamilton Slye
Bell Laboratories, Lucent Technologies
600 Mountain Avenue
Murray Hill, NJ 07974
acohen@research.bell-labs.com

Abstract

This paper describes the design, implementation and performance of a layer-7 switch which supports URL-aware redirection of HTTP traffic. Currently, there are several vendors who are beginning to announce the availability of such switches in the market, but little or no implementation and performance information is available. We discuss design issues pertaining to such switches through a prototype implementation of a URL-aware switch in the Linux kernel, and analyze the performance of our implementation. We investigate the use of TCP splicing as a mechanism for improving the performance of the switch; we explore whether TCP splicing will benefit URL-aware redirection even though HTTP connections, on average, are short-lived and transfer small amounts of data. Results from our implementation show that TCP splicing does improve the performance of URL-aware switches that handle short-lived HTTP connections. Our results also re-affirm earlier findings that TCP splicing substantially improves the performance of any application-layer proxy when large amounts of data are transferred through the splice.

1 Introduction

URL-aware redirection (also known as "content-smart switching" [ArrowPoint]) refers to the capability of a switch located in front of clients or servers to redirect HTTP requests to servers based on the URL specified by the client in its GET request. When a user enters a URL into a browser, the browser constructs an HTTP GET request which contains the URL and other HTTP client header information. With URL-aware redirection, a switch located on the path between the client and the servers will intercept the request and use the information within that request to make a decision about the server to which the request should be directed. All this happens transparently to the client.

A number of products currently available perform basic layer-4 switching [Alteon, Foundry] which involves redirecting traffic based on transport-layer (TCP) information such as TCP ports as well as network-layer (IP) information such as IP addresses. Some uses of these products are: redirecting web traffic to caches (to facilitate transparent caching), server load-balancing, and fault-tolerance. URL-aware redirection at the switch further expands the scope of information utilized by the switch to layer-7 (application) information. The benefits derived from this enhancement include the ability to direct requests for particular kinds of content (such as images or video) to servers which are optimized for delivering that content, the ability to direct requests for dynamic content to live servers instead of caches, and the ability to reduce the need for replication in environments where content is replicated among servers for load-balancing and fault-tolerance—URL-aware redirection makes it possible to use partial replication instead of full replication. Looking beyond just the URL, it is also possible to parse the "cookie" information that is carried as part of the HTTP header and make redirection decisions based on this information.

Basic layer-4 switches that provide functionality such as load balancing and support for transparent caching perform TCP traffic redirection by redirecting the initial SYN packet from the client to the chosen destination and redirecting all subsequent packets on the connection to the same destination. To accomplish such redirection, the switch needs to "peek" into the IP and TCP headers to find connection information such as IP addresses and TCP port numbers. Based on this information, the switch uses mechanisms such as NAT (network address translation) and PAT (port address translation) to redirect the connection.

Redirecting TCP traffic based on application-layer

(layer-7) information is not as simple to accomplish. For any TCP transaction, application-level information is not available until the TCP connection establishment phase has been completed. This means that connections cannot be redirected at a switch by simply peering into a SYN packet as is possible with basic layer-4 switches. The TCP connection request from the client needs to be accepted at the switch and the connection established between the client and the switch before any application-level information can be received. Once the application-level information is received, this information is parsed to determine which back-end server or cache should receive the request, and the request is redirected.

One approach for redirection is the *TCP gateway* approach where another TCP connection is established between the switch and the back-end server, the client request is passed to the server through this connection, and the response is received at the switch from the server on this connection and transferred through the other connection to the client. Another approach would be to move the switch-side endpoint of the client-switch TCP connection to the server, thereby establishing a direct TCP connection between the client and the server. This approach is a proprietary solution used by Resonate [Resonate1] and is referred to as the *TCP Connection hop* approach [Resonate2].

As far as we are aware, only ArrowPoint [ArrowPoint] and Resonate [Resonate1] provide URL-aware redirection solutions. ArrowPoint provides hardware switches (CS-100, CS-800) which are capable of performing URL-aware redirection, whereas Resonate's product (Resonate Central Dispatch) is purely a software solution. Vendors such as Foundry Networks, Alteon WebSystems and Nortel Networks together with IPivot have recently announced their intentions to deliver switches with URL-aware redirection in the near future [TechWeb1, TechWeb2]. ArrowPoint uses the TCP gateway approach to support URL-aware redirection, and we believe that other vendors that have announced products in this space will follow suit. Also, the TCP gateway approach is more general than the TCP connection hop approach as the TCP/IP stack at the back-end servers needs to be extended in the latter approach. This is impractical, for example, when the switches are used on the client side to determine whether a client request should be redirected to a nearby cache or to a remote origin server based on the content that is requested. Hence, we focus on the TCP gateway approach in this paper. In the next section, we consider URL-aware redirection in more detail and discuss the motivation for our work.

2 URL-Aware Redirection

Let us now consider a specific example of the functioning of a URL-aware switch that uses the TCP gateway approach. In this example, an HTTP request from a client to an origin server is transparently "peeked" into by a URL-aware switch, which then either forwards the request to the origin server or to a nearby cache depending on the object that is being requested. For example, if the request is for a non-cacheable item, it will be forwarded to the origin server. Normally, when a client makes an HTTP request, a TCP connection is first established with the server. The client then sends information pertaining to the object it requires as part of a GET request[1]. The server parses this request and returns the requested object to the client. When a URL-aware switch is introduced to transparently intercept client requests, the switch intercepts the connection request packets from the client (based on the destination port being port 80), and sends them up to a local application-level proxy which understands HTTP. A TCP connection is established between the client and the proxy (which now masquerades as the origin server to which the client made the request). The client then sends the GET request to the proxy, which determines the destination (in this case, the origin server or a nearby cache) based on the requested object. The proxy establishes a TCP connection to the destination, forwards the GET request on this connection, receives the response and transfers it to the client.

The TCP gateway approach exacts an overhead due to the use of two TCP connections. Data that passes from the server to the client needs to go all the way up the protocol stack to the application layer at the switch and then again down the protocol stack when it is put back on the connection to the client. To improve the performance of application-layer proxies which function as TCP gateways, TCP splicing has been proposed as a solution [Maltz1]. In this approach, once the two TCP connections are established, they are "spliced" together so that IP packets are forwarded from one endpoint to the other at the network layer without having to traverse the TCP layer to the application level on the switch. This requires that appropriate address translations and sequence number modifications be performed on

[1] There are other request tokens such as POST and HEAD, but without loss of generality we will only refer to GETs in the rest of the paper.

the packets. For example, packets arriving on the connection from the server to the switch that are to be forwarded to the client, should be translated so that the addresses and sequence numbers on these packets match the ones that would be found on the corresponding packets if the application-layer proxy received this data and then put it back on the TCP connection from the switch to the client.

In [Maltz1, Maltz3], the use of TCP splicing has been studied to improve SOCKS proxy performance. Performance numbers provided in [Maltz1] show TCP splicing to be beneficial when large amounts of data are transferred from the servers to the clients. Further, results in [Maltz2] show that HTTP caching proxy throughput can be improved with TCP splicing, again with big chunks of data transfer. HTTP caching requires data to go up the application layer to be cached anyway and may not see the full benefits of TCP splicing. Also, for a URL-aware switch that handles short-lived HTTP connections, where the number of control packets in establishing the two TCP connections may be as large as the number of data packets, it is not apparent that the benefits gained by splicing the data packets at the network layer are not offset by the extra penalty paid in snooping on the control packets to register connection state information. Hence, useful conclusions as to how a URL-aware switch will benefit from TCP splicing cannot be reached from these results. It appears that the URL-aware redirection solution from ArrowPoint makes use of TCP splicing but there is no documented data from ArrowPoint or other potential vendors whether TCP splicing benefits URL-aware redirection.

The design, implementation and performance of a hardware-based URL-aware switch called L5 are described in [Apostolopoulos]. L5 consists of programmable port controllers connected to a switch core and a general-purpose CPU. TCP splicing is used in this switch to take the CPU out of the data path once the URL is obtained and a connection to the server is established. After splicing, all packet processing is handled by the port controllers. In our software-based switch, all processing is done by the CPU since port controllers are not available. The goal of splicing in this case is to take the user-level proxy and TCP out of the data path and perform all packet processing after splicing within the kernel at the IP level.

Our contributions in this paper are as follows. First, we discuss in detail the design and implementation of a URL-aware switch in the Linux kernel that uses TCP splicing. Then, using performance measurements from our implementation, we show that TCP splicing does benefit URL-aware redirection even for small TCP sessions that transfer as few as 1 KB of data through the splice. Further, using workloads representing different object sizes and connection durations, we re-affirm the conclusions reached in [Maltz1] that when large amounts of data are transferred, TCP splicing provides substantial performance benefits. In the next section, we present implementation details of the URL-aware switch. Section 4 presents performance results from our implementation. Conclusions are presented in Section 5.

3 Implementation Details

There are four components to our switch implementation which is shown in Figure 1. The first component is an application-level proxy (*proxy-s*) which accepts TCP connections from the clients, parses the GET requests, determines the destination server and establishes a connection to that server thereby enabling URL-aware redirection. The second component is a loadable kernel module implemented inside the Linux kernel which we will refer to as the splice module (*sp-mod*). *sp-mod* monitors packets exchanged on the two connections and maintains the state machines that represent the two connections before the connections are spliced. It also maintains the state machine for the spliced connection. When appropriate, *proxy-s* communicates with *sp-mod* and indicates the pair of connections that need to be spliced. The third component is another loadable kernel module which we refer to as the NEPPI (Network Element for Programmable Packet Injection) module. This module performs low-level header manipulation operations such as network address translations and sequence number translations on the packets. When *sp-mod* receives splicing requests from *proxy-s*, it interprets those requests into low-level header manipulation instructions which it then sends to the NEPPI module. The fourth component is the Linux *ipchains* firewall, which we use to filter packets. The *sp-mod* module instructs NEPPI to filter appropriate packets to be processed. These instructions are translated by NEPPI into appropriate system calls to the *ipchains* firewall. Each of these components is described in more detail below, starting with the NEPPI and *sp-mod* modules.

3.1 The NEPPI Module

In our earlier work [Cohen], we developed a software substrate implemented as a Linux kernel mod-

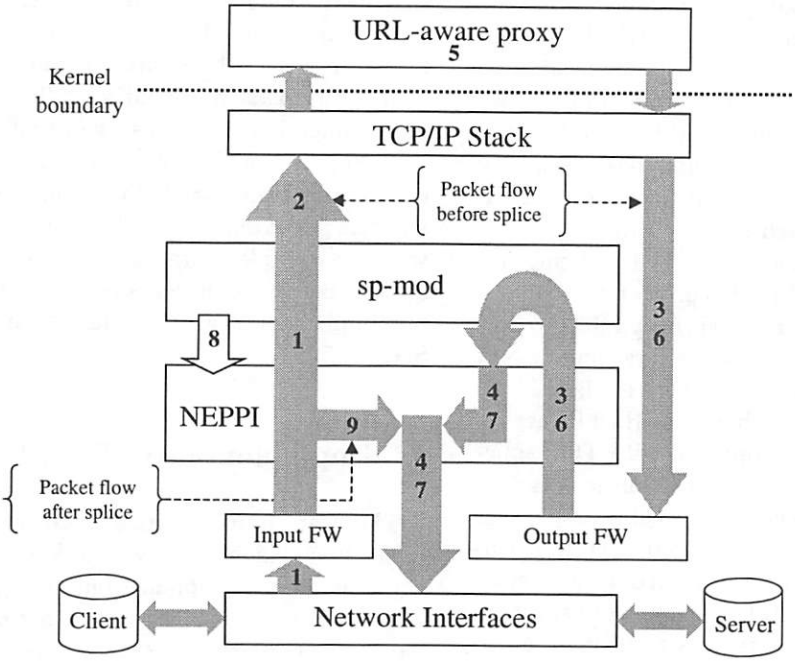


Figure 1: URL-aware switch architecture

ule (NEPPI) that provides a set of APIs which can be used to obtain, manipulate, and inject IP packets. These APIs are typically used by *gateway programs* to provide higher level services. Gateway programs can either run as user-level processes or as kernel modules themselves. Gateway programs send NEPPI rules that specify the properties of packets on which they wish to operate. Such rules may include IP address ranges for the source and destination, TCP/UDP port number ranges, etc. Based on the rules obtained from the gateway programs, NEPPI generates packet filter rules which are sent to the Linux packet filter. An arriving packet which triggers a rule is sent from the packet filter to NEPPI which either sends it to the requesting gateway program, or manipulates it in accordance with a manipulation rule specified by the requesting gateway program. Such manipulation rules include address translations, TCP sequence number changes, and TCP window size changes. Packets generated or modified by the gateway programs are returned to NEPPI, which in turn modifies them further (if they match a manipulation rule) and injects them into the network. In the case where the destination address of a packet is the switch itself, NEPPI will hand the packet to the protocol stack instead of to the network interfaces. One of our goals in designing NEPPI was to provide an architecture where packet header manipulations which are generic and which occur on a large number of packets are performed at NEPPI, whereas more specialized packet manipulations (including payload modifications) which occur only on isolated packets are performed at the gateway programs. This way, the architecture will provide a "fast path" for most packets (the ones processed by NEPPI), and a "slow path" for only a few packets (the ones processed by the gateway programs).

In the implementation of the URL-aware switch, the NEPPI module serves as the low-level packet processing substrate which is used by the *sp-mod* gateway program to perform low-level packet manipulation functions. A useful feature provided by NEPPI is support for redirecting packets to a local TCP port. Using this feature, our URL-aware switch can operate in transparent mode. Client requests destined towards origin servers are transparently intercepted and redirected to the URL-aware proxy, *proxy-s*. In this case, the *sp-mod* module requests NEPPI to redirect all incoming packets from clients to a local TCP port on the switch in which *proxy-s* is listening. NEPPI sends these packets to the local protocol stack even though the destination IP address on these packets is not that of the switch. *proxy-s*, which listens on the local TCP port, will receive this data and will even be able to determine the original destination for the data. Details on how NEPPI is used by the *sp-mod* module are given in the next section.

3.2 *sp-mod*

The *sp-mod* module monitors all packets on the client-proxy connection as well as the proxy-server connection, and maintains the TCP state machine for these connections. It also monitors all packets after the splicing is done, and maintains the state machine for the spliced connection. It accomplishes this using NEPPI as follows. The Linux *ipchains* packet filter provides INPUT and OUTPUT firewalls. The INPUT firewall filters packets from outside coming in, while the OUTPUT firewall filters packets being sent out. *sp-mod* registers with NEPPI and instructs NEPPI to forward all packets that match the following rules: **a)** all packets from clients destined to port 80 (HTTP port). **b)** all packets from servers that originate at port 80 and are destined to the URL-aware proxy. Rules **a** and **b** are translated by NEPPI into Linux system calls to the INPUT firewall. **c)** all packets generated locally (by the URL-aware proxy masquerading as the origin server) destined to the client; these packets will originate at port 80. Finally, **d)** all packets generated locally by the URL-aware proxy destined to the servers; these packets will be destined to port 80. Rules **c** and **d** will be translated by NEPPI into Linux system calls to the OUTPUT firewall.

Let us now discuss the steps taken at the switch when a client request arrives. See Figure 1 in conjunction with the following description.

When a client makes an HTTP connection request to an origin server, the SYN packet is transparently intercepted by the INPUT firewall and sent to NEPPI, which in turn forwards it to *sp-mod* **(1)**. *sp-mod* records this new connection request information and the state of this connection. It then instructs NEPPI to forward this packet to the local URL-aware proxy port **(2)**. As mentioned earlier, NEPPI is capable of forwarding packets to a local port even though the packets do not carry the switch address as the destination address. The outgoing SYN-ACK packet is intercepted at the OUTPUT firewall and forwarded to *sp-mod* **(3)**. The connection state is updated at *sp-mod* and the packet is sent through NEPPI to the client **(4)**. In order for these packets not to be intercepted again at the OUTPUT firewall, NEPPI bypasses the OUTPUT firewall when it sends packets out. Note that these packets will carry the origin server's IP address as the source address since the URL-aware proxy masquerades as the origin server. When the ACK to the SYN-ACK is received, it is again intercepted at the INPUT firewall and forwarded to *sp-mod* **(1)**, which updates connection state and forwards it to the local port **(2)**. This completes the initial handshake.

After the initial handshake, the packet(s) containing the GET request are sent by the client to the server. Again, these packet(s) are intercepted at the INPUT firewall and forwarded to *sp-mod* **(1)**. At this time, *sp-mod* records the *seq* (**seq1**) and the *ack-seq* (**ack_seq1**) on the GET packet and updates the connection state before forwarding the packets to the local port **(2)**. The URL-aware proxy processes the GET request and selects a back-end server to redirect this request based on the contents of the URL **(5)**. For example, if the switch is placed in front of clients to transparently redirect requests to caching servers, an appropriate caching server will be chosen. Once the server is chosen, it initiates a TCP connection to that server. The SYN packet from the proxy to the server is intercepted at the OUTPUT firewall and forwarded to *sp-mod* **(6)**. Again, new connection information is recorded and the packet is forwarded to the server through NEPPI **(7)**. The SYN-ACK and ACK packets similarly pass through *sp-mod* and state information is updated. Note that *sp-mod* maintains state information for all connections but cannot identify which pair of connections should be spliced. Once the initial handshake between *proxy-s* and the server is completed, the proxy sends the GET request to the server. But before that, it sends a *splice* command to *sp-mod* with the appropriate endpoint information, instructing *sp-mod* to splice the connections together. The endpoint information includes two four-tuples, one each for each connection: the client IP address and TCP port number, the address and port number of the proxy endpoint on the client side, the address and port number of the proxy endpoint on the server side, and the server address and port number. When the GET packet(s) from the proxy to the server pass through *sp-mod*, it records the *seq* (**seq2**) and the *ack-seq* (**ack_seq2**) values on the GET packet.

sp-mod uses this pair of four-tuples it received from *proxy-s*, as well as the sequence number and ack-sequence number it recorded on the two connections, for the purpose of splicing connections together. When the first ack or data packet is received from the server in response to the GET request, *sp-mod* sends instructions to NEPPI to splice the connections together **(8)**. The instructions are in the form of address translation and sequence number translation instructions. The destination address and port number on the packets from the server to the proxy are changed to those of the client, and the *seq* and *ack-seq* numbers are re-mapped to those that would be found in corresponding packets if these packets were received by *proxy-s* and

put on the TCP connection to the client. Similarly, address and sequence number translations are performed on the acknowledgment and other packets from the client before they are sent to the server. The source address and port number are changed so that the packet appears to the server as if it was sent by *proxy-s*. The *seq* and *ack-seq* numbers are re-mapped to the appropriate numbers to appear as if they were sent by *proxy-s* on its established connection to the server. The sequence number re-mapping is easily accomplished using the offsets $\delta_1 = \mathbf{seq1} - \mathbf{seq2}$ and $\delta_2 = \mathbf{ack\_seq1} - \mathbf{ack\_seq2}$, which are either added or subtracted to the sequence and ack-sequence numbers depending on the direction of packet flow.

Once the splice is done, all data packets from the server to *proxy-s*, and all acknowledgment packets from the client, are redirected to the client and server, respectively, at the network level **(9)**. As soon as the splice is performed, the two TCP endpoints on the switch bound to *proxy-s* (one on the connection to the client and one on the connection to the server) no longer receive packets and can be closed. To accomplish this, RST packets are generated by *sp-mod* masquerading as the client or the server (depending on which endpoint needs to be closed) and sent to *proxy-s*. When the data transfer has been completed, either the server or the client can initiate an active close. In either case, the FIN packets and the corresponding ACKs are still spliced so that the endpoints at the client and the server, each representing one endpoint of each connection, are also closed cleanly.

3.3 The URL-aware Proxy (*proxy-s*)

The URL-aware proxy *proxy-s* is a multi-threaded application-level program. A listener thread listens for incoming client connection requests and distributes these requests to a pool of worker threads. Once a client connection has been accepted and the GET request has been received, a connection is established with a chosen server. Before the GET request is sent to the server, *proxy-s* sends a *splice* command to *sp-mod* as described earlier. It uses the *getsockname* function call on the two socket endpoints to get the four-tuple information to be conveyed to *sp-mod*. As the client requests are transparently redirected to *proxy-s*, the *getsockname* function call on the client side socket endpoint will return the server's IP address and port number as the local endpoint. Library functions are provided which enable *proxy-s* to send the *splice* command to *sp-mod*.

The current implementation of *proxy-s* processes only the URL to make redirection decisions. The proxy can be configured to redirect requests based on extensions. For example, URLs with the GIF, JPG and HTML extensions can be redirected to different servers as shown in Figure 2. Also, CGI requests can be redirected to specified servers. A primitive form of cookie processing is available, where requests can be redirected to a server based on the presence or absence of a cookie. This will be useful, for example, when the switch is placed as a front-end to a set of caching servers. In this case, the proxy can be instructed to forward all requests that contain a cookie to the origin server instead of redirecting them to caching servers. As described in [Apostolopoulos], other HTTP request headers can be processed to provide various functionalities. As our current focus in on TCP splicing, we have not concentrated on this aspect.

3.4 *ipchains* firewall

As mentioned earlier, we use the existing Linux firewall functionality to filter packets on the *INPUT* and *OUTPUT* firewall chains. *ipchains* provides the capability to specify packet filtering rules based on source IP and destination IP address ranges, source TCP and destination TCP port number ranges, and protocol numbers. These rules are specified using system calls to the firewall. NEPPI sends commands to the firewall to filter packets requested by *sp-mod*. For each rule, a *mark* can be specified which is used by *ipchains* to tag the packets when they are filtered and forwarded. *sp-mod* uses these marks to distinguish the direction of flow for each packet so that appropriate processing can be performed. This removes the extra overhead which would be required if *sp-mod* had to look at the packet headers to determine the direction of flow.

During the connection establishment phase from the client to *proxy-s*, the packets from *proxy-s* to the client are intercepted at the OUTPUT firewall and redirected to *sp-mod*. Packets from *proxy-s* to the client are only monitored by *sp-mod* to maintain the state of the connection. It does not instruct NEPPI to perform any header modifications on these packets. This means that after *sp-mod* looked at these packets, and NEPPI received them and sent them on the network, these packets would normally pass again through the *OUTPUT* firewall, filter rules would again get fired and the packets would be forwarded back to *sp-mod*. This would create an infinite loop. A similar problem exists on the connection from *proxy-s* to the server. To get around this problem, we had to change the raw socket imple-

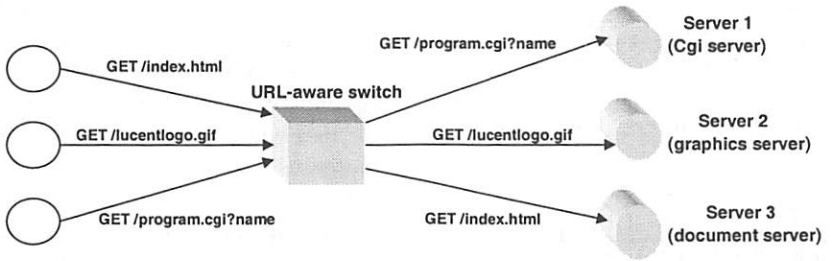


Figure 2: URL-aware redirection

mentation that NEPPI uses to send packets on the network so that the OUTPUT firewall is bypassed.

Finally, as discussed in [Maltz1], when two TCP connections are spliced, it is important that the TCP options on the two connections be adjusted and made compatible. In our current implementation, we address this problem by simply removing all options except the Maximum Segment Size option on the SYN packets on both connections.

4 Performance

We conducted three experiments to study the performance impact of TCP splicing in our Linux-based switch for URL-aware redirection. All experiments consisted of running an HTTP workload generator on five clients. The first experiment involved accessing Web sites within the Lucent Technologies corporate network; the second experiment consisted of accessing external sites, and the third experiment involved sites on the local network. For the purpose of the first two experiments, all HTTP GET requests were directed by the switch to the server specified by the client (i.e., no URL-aware redirection was performed.) The third experiment made use of the URL-aware redirection capability of the switch. We ran each workload with two versions of the proxy software running on the switch: one which performed TCP splicing (*proxy-s*), and one which did not utilize TCP splicing (we refer to this version as *proxy-ns*.) In both versions, the proxy obtains a GET request from a client, opens a TCP connection to the appropriate server, and sends the GET request to the server. At that point, *proxy-s* requests *sp-mod* (which resides in the Linux kernel) to splice the TCP connection between the proxy and the client with the TCP connection between the proxy and the server. From that point onward, *proxy-s* does not have to handle any traffic on these connections. *proxy-ns*, on the other hand, does not splice the connections and hence has to forward data between its client-side socket and its server-side socket at the application level.

We used a locally-developed HTTP workload generator called WebWatch in our experiments. This program generates multiple concurrent HTTP requests and measures the obtained performance. WebWatch reads a file containing a list of URLs and a file containing various parameters such as the desired number of concurrent requests and the number of passes through the URL file. Once WebWatch starts executing, it starts sending HTTP GET requests based on the URLs in its input file. The specified URLs are obtained along with all their embedded documents. WebWatch issues a certain number of concurrent requests (as specified in its parameters file), and then waits for the arrival of the responses or the expiration of a timer. Once all the data is received (or the timer expires), a new batch of requests is issued.

All the experiments presented here were run for a period of three minutes with a concurrency setting of 75 (i.e., 75 HTTP requests were issued at a time.) *proxy-s* and *proxy-ns* were run with a thread pool of size 30. All clients were PCs based on Pentium II 400MHz CPUs. The switch was a PC based on a Pentium III 550MHz CPU. The experiments were run multiple times to determine whether there was a significant variation among the results of different runs. The results of all runs were very similar to the results of the runs reported here.

The workload for the first experiment consisted of Web sites within the Lucent Technologies corporate network. The clients and the switch were on a Fast Ethernet network which is connected to multiple regular Ethernet networks through a router. Table 1 shows the results of this experiment. The "Conn" column shows the number of HTTP GET requests from the workload which WebWatch was able to issue within the period of the experiment. Note that the number of TCP connections for which the switch serves as an endpoint is actually twice the number which appears in the "Conn" column

Program	Conn	Util
proxy-s	13,194	1.52%
proxy-ns	7,917	5.13%

Table 1: Results for the internal workload

Program	Conn	Util
proxy-s	8,250	1.18%
proxy-ns	5,594	2.94%

Table 2: Results for the external workload

since each GET request from a client results in establishing one TCP connection between the client and the switch and another TCP connection between the switch and the server. The "Util" column shows the average CPU utilization at the switch for the period of the experiment. We see that using *proxy-s* resulted in a 67% higher number of connections than *proxy-ns,* while incurring a significantly lower CPU utilization. With *proxy-ns*, connections take longer to complete, so WebWatch was able to complete a smaller number of connections with *proxy-ns* than with *proxy-s* in the given amount of time. Recall that for each client, WebWatch was configured to submit a batch of 75 requests, and then wait for them to complete before sending the next batch. Since there is much unused CPU time, we could potentially increase the number of clients used with *proxy-ns* until the number of connections achieved matches the number obtained by *proxy-s* with just five clients. This would, however, result in an even larger difference in CPU utilization between *proxy-s* and *proxy-ns.*

The workload for the second experiment consisted of external Web sites (various news sites). Table 2 shows the results of this experiment. Again, we see that using *proxy-s* resulted in a significantly higher number of connections (47% higher) and lower CPU utilization.

To observe the impact of using TCP splicing on different file sizes, we ran a third set of experiments. The workload for this set of experiments consisted of retrieving files of a uniform size from three HTTP servers on the same Fast Ethernet network as the clients and the switch. Files had three possible extensions; the URL-aware switching capability of the switch was configured to direct each possible extension to a different server. We ran seven experiments for seven different file sizes. Table 3 shows the results of this set of experiments. File sizes are in bytes. Similar to the experiments described previously, the workload was run for a period of three minutes. The "Conn/s" column shows the average number of connections obtained per second. The table shows the significant increase in the improvement obtained by *proxy-s* when compared to *proxy-ns* as the file size increases.

proxy-s shows the biggest performance gain for large transfers. This is a result of the following: for small transfers, the overhead associated with setting up the connection and issuing the GET request is high compared to the overhead associated with transferring the data itself. This overhead is similar for *proxy-s* and *proxy-ns.* When performing large transfers, however, *proxy-ns* is strongly affected by the need to move the data between the sockets at the user-level proxy. *proxy-s*, on the other hand, does not need to handle the data at the user-level proxy at all since the two TCP connections are spliced together within the kernel.

It is important to note another advantage which *proxy-s* has when compared to *proxy-ns.* After the splicing of the two TCP connections (client to proxy and proxy to server) is accomplished, the connections are reset, so they cease to exist. When no splicing is used (i.e., in the case of *proxy-ns*), the connections exist for the duration of the data transfer and for an additional period beyond the time when the connections are closed (the 2MSL timeout required by TCP). This results in a much larger number of TCP connections with state in the protocol stack when running *proxy-ns.*

As seen in Table 3, the CPU in the switch is fully loaded when running *proxy-ns.* We were not able to fully load the CPU when running *proxy-s.* For files of size 10,000 bytes and larger, the 100 Mbit/sec network was fully utilized. For smaller files, neither the network bandwidth nor the CPU utilization was a bottleneck. Hence, one might expect to obtain larger numbers of connections for those files than the numbers we obtained. We are not certain about the reasons for not obtaining larger numbers. Possibly, a limitation of the servers was reached, or perhaps some limitation of the Linux TCP/IP stack was reached. We plan to investigate this issue further in the future.

Empirical data indicates that the average web object size is around 10 KB [Lee]. For objects of this size, Table 3 shows that TCP splicing results in a 53% increase in the number of connections along with a 38% decrease in CPU utilization.

File size	Splice *(proxy-s)*			No splice *(proxy-ns)*		
	Conn	Util	Conn/s	Conn	Util	Conn/s
1,000	233,035	61.32%	1,295	183,307	95.49%	1,018
2,000	227,107	64.85%	1,262	172,984	93.61%	961
5,000	214,782	71.20%	1,193	153,202	97.99%	851
10,000	177,216	61.84%	985	116,071	99.82%	645
20,000	89,477	21.24%	497	82,866	99.56%	460
30,000	62,907	13.42%	349	60,939	96.80%	339
300,000	6,843	1.60%	38	5,455	99.45%	30

Table 3: Results for a uniform workload of different file sizes

5 Conclusions

The implementation of a URL-aware redirection switch was presented. The switching functionality is provided by loadable kernel modules for the Linux OS along with a user-level proxy. The user-level proxy serves the purpose of accepting connections from clients, determining the requested URLs, making decisions about the appropriate servers, and submitting the requests to the servers. Upon submission of a request to a server, the proxy removes itself from the data path by requesting the kernel to splice at the IP level the connection between the client and the proxy with the connection between the proxy and the server [Maltz1, Maltz2]. The performance impact of TCP splicing was studied by comparing the performance of the switch with and without splicing on a variety of workloads. TCP splicing resulted in a significant performance improvement on all workloads. As expected, the effects of TCP splicing are particularly striking for large requests, but our results show significant performance gains even for workloads consisting of small requests of one kilobyte.

Acknowledgments: We would like to thank Reinhard Klemm for providing us with his HTTP workload generator WebWatch.

References

[Alteon] ACEdirector, *http://www.alteon.com/products.*

[Apostolopoulos] Apostolopoulos, G., V. Peris, P. Pradhan, and D. Saha. *L5: A Self Learning Layer-5 Switch.* IBM Research Report 21461, 1999.

[ArrowPoint] "The content smart internet", *http://www.arrowpoint.com/solutions/whitepapers/CSI.asp.*

[Cohen] Cohen, A., and S. Rangarajan. A Programming Interface for Supporting IP Traffic Processing. In: *Proceedings of the 1st International Working Conference on Active Networks (Lecture Notes in Computer Science, Vol. 1653, Springer),* pp. 132–143, 1999.

[Foundry] "ServerIron server load balancing switch", *http://www.foundrynet.com/serveriron sepc.html.*

[Lee] Lee, R., and G. Tomlinson. Workload Requirements for a Very High-Capacity Proxy Cache Design. In: *Proc. of the 4th International Web Caching Workshop (NLANR/CAIDA),* 1999.

[Maltz1] Maltz, D.A., and P. Bhagwat. *TCP Splicing for Application Layer Proxy Performance.* IBM Research Report RC 21139, 1998.

[Maltz2] D.A. Maltz and P. Bhagwat. *Improving HTTP Caching Proxy Performance with TCP Tap.* IBM Research Report RC 21147, 1998.

[Maltz3] D.A. Maltz and P. Bhagwat. MSOCKS: An Architecture for Transport Layer Mobility. In: *Proc. of IEEE INFOCOM '98,* pp. 1037–1045, 1998.

[Resonate1] "Resonate Products—Central Dispatch", *http://www.resonate.com.*

[Resonate2] "Resonate Central Dispatch TCP Connection Hop", *http://www.resonate.com/products/tech_overviews.html.*

[TechWeb1] "Switch Vendors Detail Enhancements", *http://www.techweb.com/wire/story/TWB19990316S0006,* 1999.

[TechWeb2] "Portals Provide Layer 4 Acid Test", *http://www.techweb.com/wire/story/TWB19990315S0009,* 1999.

Prefetching Hyperlinks

Dan Duchamp

AT&T Labs – Research

Abstract

This paper develops a new method for prefetching Web pages into the client cache. Clients send reference information to Web servers, which aggregate the reference information in near-real-time and then disperse the aggregated information to all clients, piggybacked on GET responses. The information indicates how often hyperlink URLs embedded in pages have been previously accessed relative to the embedding page. Based on knowledge about which hyperlinks are generally popular, clients initiate prefetching of the hyperlinks and their embedded images according to any algorithm they prefer. Both client and server may cap the prefetching mechanism's space overhead and waste of network resources due to speculation. The result of these differences is improved prefetching: lower client latency (by 52.3%) and less wasted network bandwidth (24.0%).

1 Introduction

The idea of prefetching Web pages has surely occurred to many people as they used their browsers. It often takes "too long" to load and display a requested page, and thereafter several seconds often elapse before the user's next request. It is natural to wonder if the substantial time between two consecutive requests could be used to anticipate and prefetch the second request.

The related work section of this paper cites 14 distinct prior studies of prefetching for the Web. In each of these studies, any *transparent* prefetching algorithm (meaning that the user is uninvolved) is also *speculative*. Speculative means that some system component makes a guess about a user's future page references based on some knowledge of past references, gathered from that user alone or from many users.

This paper examines a new method for prefetching Web pages into the client cache that is also transparent and speculative. While the basic approach to prefetching is the same in all studies, major differences among prefetching methods lie in the details. The major characteristics that distinguish this study from prior ones are:

1. We have implemented our ideas, twice in fact.
2. Clients send reference information to servers, which then disperse aggregated information to other clients in near-real-time. The reference information indicates how often hyperlink URLs embedded in pages have been previously accessed relative to the embedding page.
3. Servers aggregate the reference information in near-real-time rather than, say, overnight, allowing for prefetching decisions based on up-to-date usage patterns.
4. Clients initiate prefetching according to any algorithm they prefer; they also control how to age reference information.
5. Prefetching is not limited to URLs on the same server, or to URLs previously accessed by the same client.
6. Many un-cacheable pages may be prefetched, including pages generated dynamically, by query URLs, or those having cookies.
7. Both client and server can cap the prefetching mechanism's overhead and waste.
8. In one implementation, proxies are used to avoid changing either the browser or Web server.
9. HTTP is extended.
10. The prefetching algorithm continually measures bandwidth available to the client and limits prefetching requests to a fraction of the available bandwidth.

Most of these characteristics are not shared by most prior studies.

The result of these differences is improved prefetching: lower client latency (52.3% reduction) and much less waste (62.5% of prefetched pages are eventually used).

Section 1.1 gives a brief sketch of how prefetching works. Section 2 summarizes prior work in the area. Section 3 describes some conclusions, drawn from reference

traces, that support certain design decisions, and Section 4 describes the design as well as two implementations. Section 5.1 analyzes the costs and benefits of this approach to prefetching.

1.1 Summary of Prefetching Method

Upon their first contact in "a while," a client and server negotiate the terms of prefetching: whether it will happen, and when and how much information they will exchange to support it.

Once terms have been negotiated, clients send *usage reports* to servers promptly but as part of the critical path of a GET request. Usage reports describe the fact that one or more URLs embedded within a page was recently referenced from that page. For example, if a client references page P and then references page Q based on an HREF embedded in P, then the usage report will indicate that P referenced Q; the usage report will also include other information useful for prefetching, such as the size of Q and all its embedded images, and how much time elapsed between the reference to P and the reference to Q.

The server makes a best effort to accumulate the information from all usage reports that pertain to the same page, P; the usage reports are kept ordered by time. Whenever the server delivers P to a prefetch-enabled client, it attaches a summary (called a *usage profile*) of the information that it has obtained from earlier usage reports for that page, from all clients. The summary, whose format was negotiated earlier, indicates how often HREFs embedded in page P have been referenced, relative to the number of references to P in the same time period(s). Time is measured by references; thus, for example, a client can negotiate to receive usage profiles that describe the references of embedded HREFs relative to the last 10, 25, and 50 references of the page P.

A client that receives a usage profile along with page P may choose whether or not to prefetch any HREFs embedded in P, according to any algorithm it prefers. The decisions whether and how to prefetch rest with the client because the client best knows its own usage patterns, the state of its cache, and the effective bandwidth of its link to the Internet.

2 Related Work

Here we survey 14 prior separate efforts relevant to prefetching in the Web, divided into three categories: software systems; papers describing algorithms, simulations, and/or prototypes; and papers that establish bounds.

2.1 Software Systems

Smiley is similar to our work in that prefetching is decided by the client based on usage statistics about embedded HREFs. The client also continually monitors its available bandwidth. One major difference is that images embedded within prefetched pages are not themselves prefetched. The Smiley implementation is only a demonstration, and results [16, 17] are obtained by a simulation study of accesses to two frequently accessed pages at UCLA.

Major differences in the gathering and use of usage statistics between Smiley and our work are that there is no method for clients to inform servers of their usage patterns, and that usage statistics are gathered not in real time but over many days and then they are not aged.

Although Smiley does not include a method for clients to inform servers of their usage patterns, client-side and server-side observations of usage patterns are merged in the simulation. Jiang and Kleinrock conclude that using server-side statistics in combination with those of the client yields a higher hit rate than using client-side statistics alone. A basic hypothesis of our work is that an individual client can prefetch accurately based upon usage statistics gathered from a large population. The Smiley results suggest that our hypothesis is sound.

Wcol [5] is a research prototype available on the Web. It prefetches embedded hyperlinks top-to-bottom without regard to likelihood of use. Embedded images of prefetched pages are also prefetched. Bandwidth waste can be capped by configuring Wcol to prefetch no more than a certain number of hyperlinks, and no more than a certain number of images embedded within prefetched hyperlinks.

PeakJet2000 [26] is the second major version of a commercial product. PeakJet runs on the client machine. It maintains a separate cache and provides a set of tools for speeding Web access, some of which require user action. True prefetching exists in two modes, "history-based" and "link-based." The user picks the mode. History-based prefetching prefetches an embedded link only if that client has used it before (i.e., it performs an IMS GET of a cached page). Link-based prefetching prefetches all embedded HREFs.

NetAccelerator 2.0 [24] is a similar commercial product for Windows clients. Unlike PeakJet, it prefetches into the browser's disk cache. Its prefetching algorithm is the same as PeakJet's link-based prefetching: all hyperlinks and their embedded images are prefetched.

2.2 Algorithms and Simulations

The method proposed by **Bestavros** in [2] is that "a server responds to a clients' [sic] request by sending, in addition

to the data (documents) requested, a number of other documents that it speculates will be requested by that client in the near future."

A Markov algorithm is used to determine related "documents." The server examines the reference pattern of each client separately. Two documents are considered related if the client has accessed them in the past within a certain time interval. When a document is fetched, the server also pushes to the client any other document that is transitively related to it and whose likelihood of use is greater than a threshold value.

Usage statistics are gathered over 60 days, updated daily. Documents are considered related if requests from same client come within 5 seconds of each other. Among the results of this study are (1) that more recent usage statistics yield better results, as does more frequent updating of the "related" relation; (2) and "speculation is most effective when done conservatively." That is, with each incremental decrease in client latency, the extra bandwidth consumed and extra load imposed upon the server becomes greater.

The main point of the **Dynamic Documents** prototype [18] was to investigate how implementing documents as programs rather than static files might provide a means to help mobile clients adjust to enormous variations in bandwidth. Prefetching was added more as demonstration of a possibility rather than as a serious proposal. Pages in the history list are prefetched, with the result that bandwidth use is increased approximately 50% but only 2% of prefetched pages are used.

In the work of **Padmanabhan and Mogul** [25], a server maintains per-client usage statistics and determines related-ness through a graph-based Markov model similar to that of Bestavros. The graph contains a node for "every file that has ever been accessed" and is updated off-line, nightly for example. Related-ness is determined by edges through the nodes weighted by the probability that one will be accessed soon after the other. Whereas Bestavros defines "soon" by an amount of time, Padmanabhan and Mogul define it by a number of accesses from the client that occur in between.

When a GET is serviced, the server calculates a list of its pages that are likely to be requested in the near future, using some probability threshold. This list is appended to the GET response, and the client decides whether to actually prefetch. Another HTTP extension allows the client to indicate to the server that a certain GET is a prefetch, so that the server will not recursively compute related-ness for the prefetched page.

Trace-driven simulations show that average access time can be reduced approximately 40%, at the cost of much increased network traffic (70%). Another result suggests that prefetching is more beneficial than increasing bandwidth. That is, when prefetching causes a 20% increase in traffic, the latency is lower than it would be without prefetching but with 20% extra bandwidth. A third result is that prefetching increases access time variability, but very little.

The basic idea of **Top10** [21] is for servers (and proxies) to publish their most-accessed pages ("Top 10"). Downstream components (clients and proxies) prefetch some fraction of the list. The approach is parameterized two ways. One parameter indicates how many times a client must have contacted a server before it will prefetch at all. The other parameter indicates the maximum number of pages it will prefetch from a server. Results are by trace-driven simulation with traces from 5 sites. As more pages are prefetched, the percent of prefetched pages that are eventually used rises quickly and levels off at between 3% and 23%, depending on the trace. This suggests that the size of "TopN" should be small.

Fan et al. [13] evaluates several techniques for reducing client requests and observed latency. The evaluation is based mostly on trace-driven simulations of dialup users [15]. The authors have also implemented their prefetching ideas using CERN `httpd`. Pages are prefetched into a simulated browser cache only from a shared proxy cache, never from servers, so no extra wide-area traffic is generated. Consequently, prefetching is limited by the degree to which one client is expected to use a page that it or some other client sharing the same proxy has used in the past, and which is still in the proxy's cache.

Simulation results indicate that (perfect) prefetching and delta-compression [22] reduce latency considerably more than either HTML compression or merely increasing the size of the browser's cache. Using all three techniques with a finite browser cache resulted in only 30.3% latency reduction. However, prefetching reduced the number of client requests by 50%.[1] The 50% request savings, in turn, is limited by the fact that pages are prefetched only from the proxy.

Image objects were prefetched more often than HTML objects (64-74% versus 13-18%), and the prediction accuracy was higher for image objects (approximately 65% for JPEG and 58% for GIF versus 35% for HTML and "other" types). One possible explanation for this pattern is that this work uses a Markov-model prediction algorithm. Such algorithms view objects as independent, and often simply re-discover which images are embedded in an HTML page.

In the implementation, there is a proxy on the client side and one on the modem side. As in our work, the path of a request is browser to client-side proxy to modem-side proxy to server. Also like our work, The client-side proxy apparently piggybacks hit info on requests. After

[1] Latency savings are less than request savings because cached pages — those that can be prefetched — tended to be smaller than pages that had to be fetched from servers.

the modem-side proxy processes a request, it keeps the connection open, generates a list of URLs to prefetch, and "pushes" them into the client-side proxy's cache. The proxy prefetches only items in its cache. The predictor at the modem-side proxy remembers the client's last few requests but does not know the state of its cache, resulting in possible duplication.

Cohen and Kaplan [6] investigate three other types of prefetching: opening an HTTP connection to a server in advance of its (possible) use (pre-connecting); resolving a server's name to an IP address in advance of opening a connection to the server (pre-resolving); and sending a dummy request (such as a HEAD) to the server in advance of the first real request (pre-warming). Pre-connecting is motivated by the substantial overhead of TCP connection establishment. Pre-resolving is a subset of pre-connecting: the only part of a GET request done in advance is to translate the server's name into an IP address. Pre-warming is a superset of pre-connecting: its purpose is to force the server to perform one-time access control checking in advance of demand requests. In a trace-driven simulation, the three techniques reduced the number of "session starting" HTTP interactions whose latency exceeded 4 seconds from 7% to 4.5%, 2%, and 1%, respectively. This work represents a more conservative approach to prefetching than our own: much less complex, more likely to work without unintended consequences, and less capable of reducing latency.

Cunha's work [11] presents a very simple browser prefetch mechanism plus two mathematical models taken from prior work on other topics, that are used to indicate whether the mechanism should be invoked. His dissertation [12] provides additional detail not supplied in the paper, including recognition of the complications of file size and the need to age the usage information.

The prefetch model is that only the client is active in gathering usage information and making prefetch decisions. Hence, a user prefetches only pages that he has accessed before. The earlier references resulted in Markov chains with three types of links indicating whether objects that were accessed within a time window are unrelated, related by one being embedded in the other, or related simply by being likely to be accessed at about the same time.

The mathematical models – based on DRAM caches and linear predictive coding for speech processing – attempt to classify a user's behavior as "surfing" or "conservative." Surfing behavior references many different URLs whereas conservative behavior frequently re-references the same URLs. Very high hit rates (e.g., over 80%) are possible when the user's behavior fits the model.

2.3 Bounds

One of the results of the **Coolist** papers [27, 28] is a mathematical analysis of how accurate prefetch predictions must be (or, alternatively, how lightly loaded the network must be) in order for prefetches not to interfere with demand fetches and thereby *increase* the average latency of all fetches. The formula is $R < E/(1+E)$, where R is network utilization without prefetching and E is the ratio of hit rate with prefetching to traffic increase caused by prefetching. The papers also present a taxonomy of prefetching approaches that range from conservative to aggressive in their use of bandwidth, and "Coolist," a prefetcher that allows the user to choose the level of aggressiveness for prefetching user-specified groups of pages.

The surprising conclusion of **Crovella and Barford**'s trace-driven simulations [9, 10] is that prefetching makes traffic burstier and thereby worsens queueing. This is surprising because, generally speaking, a side effect of prefetching is to smooth short "spikes" of demand fetching into longer "trickles" of prefetching. The explanation lies in the definition of prefetching: at the start of a session *all* the files to be accessed in that session are prefetched immediately, creating an initial burst of demand. Crovella and Barford provide a solution, "rate-controlled prefetching," that smoothes out traffic. Rate-controlled prefetching approximates realistic prefetching, where the lookahead is limited. The analysis of rate-controlled prefetching also uses a better definition of prefetching – pages are prefetched one at a time and not with perfect accuracy. Such rate-controlled prefetching significantly reduces queue length over a wide range of prefetch accuracies.

An important study by **Kroeger et al.** [19] establishes bounds on the latency reduction achievable through caching and prefetching, under idealized conditions. Their most widely noted conclusion is that, even employing an unlimited cache and a prefetch algorithm that knows the future, at most 60% latency reduction can be achieved.

However, this study assumes (1) that "query events and events with `cgi-bin` in their URL cannot be either prefetched or cached" and (2) prefetching will always miss on the first access to a particular server. Neither is true in our work. The first assumption does not apply to our work because it is appropriate for a cache shared by several users, but our algorithm prefetches into a user-specific cache. Because many URLs represent queries or dynamic content (16.3% as shown in Section 3.2), removing this assumption in particular could yield an upper bound significantly above 60%.

3 Preliminary Experiments

This section describes preliminary experiments that were undertaken to reduce the number of unsupported assumptions and design decisions, and to test the extent to which pessimistic conclusions of certain earlier studies [4, 19] apply to this study.

Unfortunately, the data needed for the experiments described in this section is not present in well known existing logs such those from DEC [8], Boston University, Berkeley [15], and AT&T. Accordingly, we had to gather our own logs. A separate trace was gathered at AT&T Labs over several months in 1999. A "snooping" proxy produced, for every GET request, the following information: requesting client; URL; IMS request or not; `Referer` field, if present; time request was received by snooper; time first response byte was received; time last response byte was received; status code; Content-type field; lengths of header and content; time to expiration, and how computed; the number of embedded URLs; and whether the response would be cached and, if not, why not. In addition, all pages were permanently logged, including in cases where an unaltered proxy would not have cached it. It was necessary to log content because some experiments described in this section need it. For example, Experiment E determines, among other things, whether the URL of a referenced page is embedded as a hyperlink in any page accessed within the previous 30 seconds.

The trace consists of 92,518 references generated by members of the author's department (6 clients) over a span of about 5 months. All clients were attached to the same high speed LAN at 10Mb/sec. The LAN is attached to the Internet via a partial DS3. Drawing traces from a high speed environment is desirable because inter-reference times are likely to reflect the user's actual think time rather than bandwidth limitations of the local environment.

3.1 Cross-Server Links

Experiments A and B, respectively, address the assumptions in [19] that "prefetching can only begin after the client's first contact with that server" and that "query events and events with *cgi-bin* in their URL cannot be either prefetched or cached." Neither prohibition applies to this work. How significant are the effects of lifting these prohibitions?

Experiment A. The experimental question is: in those cases where one page refers to another, what fraction of referenced links (and bytes) are from sites different from the site of the referring page?

Analysis was done by parsing site information from the URLs. Sites are deemed to be different if both their names and IP addresses differ at the moment that the log analysis program runs. In some cases, months elapsed between the gathering of log data and the last run of the analysis program. It is assumed that during that time very few host pairs flip-flop between being same and different.

The results are that 28.9% of referenced hyperlinks, representing 19.9% of bytes, are to URLs on other servers. This seems high, and might reflect references skewed to commercial sites packed with advertisements on other sites.

3.2 Non-cacheable Links

Prefetch-ability in this work is not the same as cacheability in the literature. The reason is that the prefetcher brings pages into a *client-specific* cache. Because the cache is not shared, it is possible to cache cookies, query URLs and dynamic content. Studies of techniques for shared caches exclude such pages, with considerable effect. For example, one study [14] found that, in one sample, 43.1% of documents were uncacheable, mostly because of cookies (30.2%), query URLs (9.9%), obvious dynamic content (5.4%), and explicit cache-control prohibitions (9.1%).[2]

Experiment B. The experimental question is: what fraction of referenced links (and bytes) are query URLs or "obvious" dynamic content?

The definition of "obvious dynamic content" is derived from the one most often seen in the literature: a URL is assumed to specify dynamic content if it contains `cgi`. This heuristic is outmoded, as there are ever more tools for producing dynamic content, and these tools produce URLs by a variety of conventions, not just `cgi-bin`. The effect of using an old heuristic is that the proportion of dynamic content is underestimated, meaning that we err on the conservative side.

The results are that 16.3% of referenced hyperlinks, representing 18.2% of bytes, are to query URLs or to URLs containing `cgi`.

The results of experiments A and B are not meant to challenge or invalidate conclusions such as those in [19], because the bounds in that paper were developed under conditions that are unrealistically favorable. These results are meant to show merely that the theoretical limit to latency improvement via prefetching is probably higher than 60%, for two reasons. First, [19] assumes a shared cache and therefore does not prefetch certain pages that can be prefetching into a private cache. Second, [19] excludes the possibility of prefetching from a "new" site.

[2] The individual numbers do not add to 43.1% because some documents are uncacheable for several reasons.

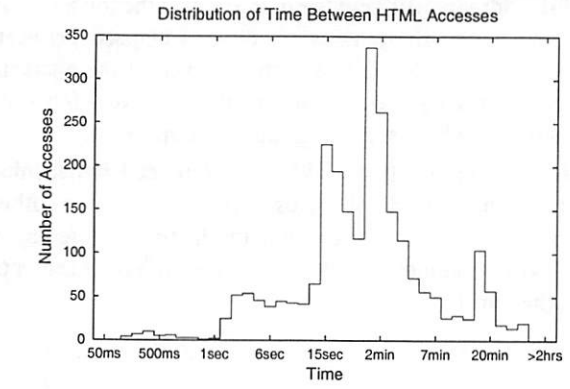


Figure 1: Time Between Referer and Referee (HTML)

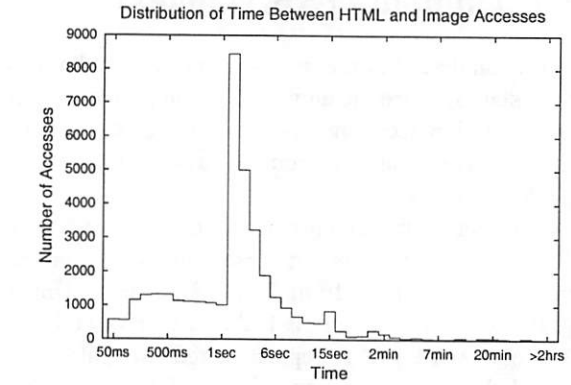


Figure 2: Time Between Referer and Referee (Image)

3.3 Inter-reference Time

Prefetching is not a purely algorithmic problem. Even given an oracle that could predict future accesses perfectly, prefetching results might be imperfect if a page were demanded during the time between the prediction of its need and its arrival. In such a case prefetching might still lower the observed latency, but conventionally such partial success is counted as a prefetching failure.

Experiment C1 determines the distribution of the time interval between referring and referred-to pages. Since Web data expires after a while, another timing issue is the distribution of expiration intervals. Because prefetched data can expire before it is demanded, prefetching too far in advance is also a problem. Experiment C2 determines the distribution of expiration times.

Experiment C1. The experimental question is: in those cases where one page refers to another, what is the distribution of the time between (1) the end of the transfer of the referer and (2) the beginning of the transfer of the referee? This is the amount of time available to the prefetcher.

The results are broken down into two categories: HTML referencing HTML (Figure 1), and HTML content referencing image content (Figure 1). The breakdown is intended to capture the difference between references to embedded hyperlinks and images, respectively. As shown, the browser-generated references to embedded images happen much more quickly than user-generated references to hyperlinks. The median inter-reference time between two HTML pages is 52 seconds, while inter-reference time between an HTML page and an image page is 2.25 seconds. However, a significant fraction of images (32.3%) are requested within one second or less of the referencing HTML page. This suggests that when an HTML page is prefetched, its embedded images should be prefetched too. Doing so substantially increases the penalty for being wrong.

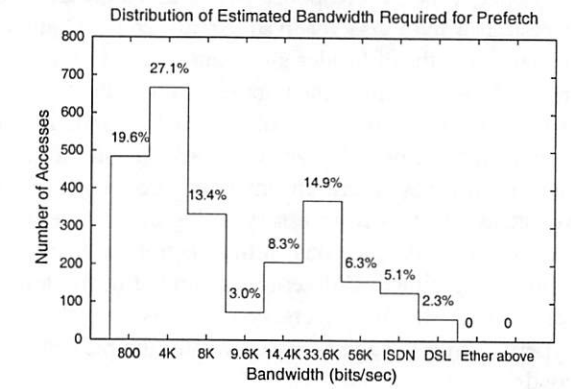


Figure 3: Bandwidth Required for Prefetching One Page

Figure 3 is the distribution of "total page size" divided by inter-reference time. That is, references in the log were analyzed to determine the size of an HTML page plus all its embedded images, whether or not they were actually referenced – an overestimate of what is needed to completely prefetch a page. Then this number was divided by the time between when that page's referrer was loaded and when the page was requested. This quotient is the maximum amount of data a prefetcher would have to deliver divided by the amount of time available to it: the bandwidth needed to prefetch one page. The median bandwidth is around 5KB/sec, safely within the capacity of even a dialup modem, suggesting that several pages could be prefetched while remaining within bandwidth constraints.

Experiment C2. The experimental question is: what is the distribution of expiration times, and how are the times computed? Prefetching can be harmful if prefetched content expires before it is demanded.

As Figure 4 shows, the most common expiration time

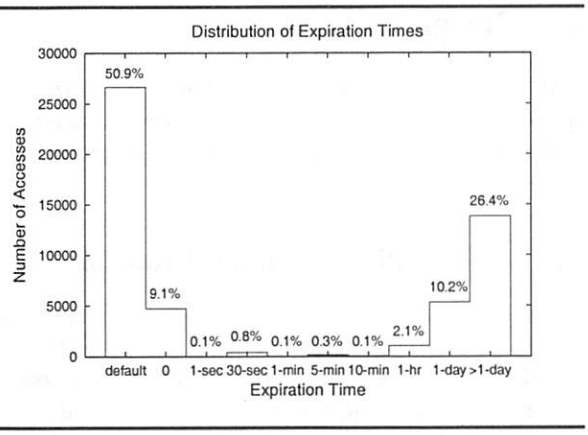


Figure 4: Time Until Expiration

is zero (60.0%). Typically (50.9%), zero is determined by default because the server has provided no information (`Expires` or `Last-modified` headers) upon which to base an estimate. The rest of the time (9.1%), the server has provided identical `Expires` and `Date` headers.

Strictly speaking, a zero expiration time should doom prefetching because prefetched content will be expired once it is demanded from the cache. However, since the meaning of a zero expiration time is "use only once," in our implementation we take the following steps. First, as explained in Section 4.1, a special header (`Prefetch: Prefetch`) is present in prefetched pages. The cache freshness-check code has been altered to ignore an explicit zero expiration (i.e., Expires = Date) when such a page has the prefetch header. When a such a page is eventually read, the cached page is re-written with the current time in the expires field and the `Prefetch: Prefetch` header overwritten.

The median non-zero expiration time (more than one day) far exceeds the average inter-reference time (52 seconds), indicating that too-early prefetching is not a practical concern. The median expiration time is so large because in many cases expiration is based only upon `Last-modified` headers, and many pages have not been modified in months.

3.4 The Naive Approach

The naive approach to prefetching based on embedded links [5, 26, 24] is to prefetch all embedded links or some number of them, without regard to the likelihood of use. Experiment D determines that some pages can have a very large number of embedded links, representing many bytes.

Experiment D. The experimental question is: what is the distribution of the number of links per page, how many bytes do these links represent, and what fraction of links/bytes are actually accessed?

As the snooper recorded the content of each page, it also parsed the pages and logged each embedded URL and its reference type (HREF, IMG SRC, etc.). The log analyzer determined which of these URLs were later accessed from that page. It also determined the size of each embedded HREF URL and its embedded images by calling a utility similar to the popular `webget`. Therefore, the sizes were determined, in some cases, weeks or months after the fact. It is assumed that size changes occurring the interim are negligible, or at least not substantially skewed in one direction or another.

The average number of links per page is 22.6. The average size of these links is 7760 bytes. The fraction of links and bytes accessed is very low (5.4% and 3.8%, respectively), again perhaps because of commercial portal sites that are packed with links. These results reinforce the importance of accurate prefetch predictions.

3.5 Markov Prediction Algorithms

Some prior work (e.g., [2, 13, 25]) has used Markov modeling to make reference predictions. A Markov prediction algorithm makes no use of structure information such as the `Referer` field, but instead regards earlier references as an unstructured string drawn from the "alphabet" of page IDs and attempts to discover patterns in the string. Recurrence of the initial part of a detected pattern triggers prefetching of the remainder of the pattern string. Markov algorithms seek patterns within a "window" of prior references, where the window typically is measured by time or number of references.

The two advantages of the Markov approach are that it does not depend upon the `Referer` field (which may not always be present) and that it can discover non-intuitive patterns. However, there are significant disadvantages to Markov algorithms. One is that such algorithms often have a high cost, measured in storage and compute time. The reason is that a Markov algorithm typically represents pages as graph nodes and accesses as weighted graph edges. With no structure to guide the search for patterns, predictive ability is improved by enlarging the window: retaining as many nodes as possible and searching as many paths as possible. Second, naive implementations of Markov algorithms have trouble aging their data. We hypothesize that Markov algorithms suffer a third drawback of being too general – they discover patterns that, to a large degree, are already obvious in the HTML of recently accessed pages and might be more simply extracted therefrom.

Experiment E compares the fraction of Markov "dependencies" among pages (that is, patterns where an access to page X tends to be closely followed by an access to page Y) that are also embedded links within recently ref-

Time (sec)	10	30	60	120
Embedded	70.0%	74.6%	76.0%	77.2%
References	1	3	5	10
Embedded	39.6%	66.3%	73.2%	78.3%

Figure 5: Percent of Markov Dependencies Also Present as Embedded Links Within Recently Accessed HTML Pages

erenced HTML pages.

Experiment E. The experimental question is: what fraction of "dependent" pages/bytes might also be easily detectable as embedded links? Two definitions of dependency are taken from much-cited works, [2] and [25]. Two references represent a dependency if they are made by the same client within a certain time or a certain number of prior references, respectively. As explained below, only a certain class of dependencies is considered. The results are shown in Figure 5.

The experiment computed, for every URL accessed, whether that URL was present as an embedded link within an HTML page that was previously referenced either within a certain time interval (10, 30, 60, and 120 seconds), or within a certain number of prior HTML pages (1, 3, 5, and 10). For each case, the table reports the fraction of URLs that were present as embedded links in those earlier HTML pages.

The interpretation of these results is not straightforward. First, there is no single "Markov algorithm" to compare to. For example, a Markov algorithm will declare two references to be dependent if they occur consecutively *with more than a certain probability*. Varying the threshold probability yields different algorithms. Second, Markov-based algorithms can find dependencies between any two pages, not just between an HTML page and another page. Therefore, Experiment E sheds some light on Markov algorithms versus our approach of examining links embedded in HTML, but it is not a definitive evaluation.

Experiment E considers only dependencies $X \rightarrow Y$, where X is an HTML page. In this subset of dependencies that a Markov algorithm might find, the results indicate that the (presumably) simpler method of inspecting HTML can locate a high percentage of dependencies. For instance, 39.6% of such dependencies can be discovered by inspecting only the single prior HTML page; 70.0% can be discovered by inspecting the HTML pages that were referenced in the preceding 10 seconds. These results are conservative, since it is not certain that any particular Markov algorithm would succeed in locating 100% of the dependencies.

4 Design

This section adds to Section 1 by explaining two topics in greater depth: the protocol used to exchange usage reports and usage profiles and the prefetching algorithm used by the client.

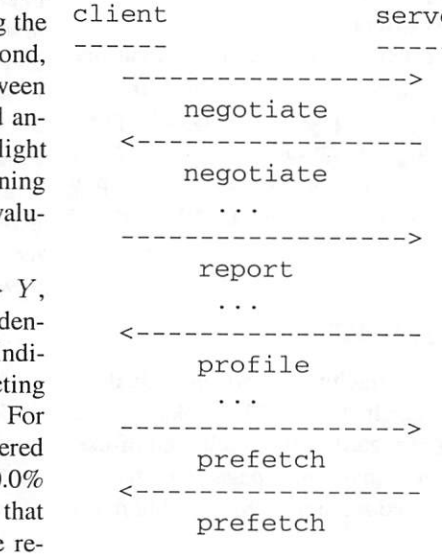
4.1 Information Exchange Protocol

One new HTTP header is defined, `Prefetch`. There are five Prefetch directives: `Negotiate`, `Report`, `Profile`, `Prefetch`, and `Halt`. Some directives take arguments. In particular, usage reports are provided as arguments to the `Report` directive and usage profiles are provided as arguments to the `Profile` directive.

Both client and server can limit the amount and define the format of prefetching information they exchange, and specify when such exchanges may take place. When initiated by a client, `Prefetch: Negotiate` should be sent along with a message to which a response is expected, such as the first GET. `Prefetch: Report` may be sent by the client along with any message after negotiation has finished. `Prefetch: Profile` is sent by the server only on GET responses, since the profile pertains to only those URLs whose body is included in the message.

The `Negotiate` directive can be specialized through a number of arguments that specify whether usage reports will be sent occasionally, periodically, after a certain number of page accesses, or once the usage report reaches a certain size.

The most common use of `Negotiate`, `Report`, `Profile`, and `Prefetch` directives is as follows:



Three other arguments to `Negotiate` specify the desired format of future usage profile directives. One argument indicates how to age the data. For example,

`last=10,20,50` means that the server's usage profile should, if possible, summarize which embedded references have been used in the last 10, 20, and 50 references to the page. The server is bound only to make a best effort to retain enough data to deliver usage profiles in the format it has negotiated. The other two arguments specify limits on the size of the usage profile: absolute byte count and relative to the attached GET response.

4.1.1 Record Format

A usage report includes the referring and referred-to URLs, when the request took place, how long it took to satisfy, and the size of the result. Only successful references to HREFs are reported.

Usage profiles comprise two sorts of records. One is `Last N` where N is an integer. The other sort of record is similar to a usage report prefaced by an integer, M. The two types of records are intermixed, with one `Last N` record followed by some number of "`M...`" records. This pattern repeats. The meaning of such a sequence of records is that the last N times this page was referenced, it happened M_i times that URL_i was referenced by that page. The sum of M_i need not equal N.

Auxiliary information, such as the elapsed time between references to P and some embedded HREF Q, appear in the summary as medians plus standard deviations.

4.2 Prefetching Algorithm

The client-side prefetch algorithm that has been used for the measurements in this paper is the following. During negotiation with a server, the client asks for the usage profile to summarize the last 10 and 50 references. The client continually measures the speed of its HTTP GET transfers, and maintains a running average; the speed varies depending on many factors so the data is inaccurate, but inaccurate data is regarded as better than no data. Whenever a page is demand-fetched, its embedded HREFs are noted during the parsing necessary to display it. These HREFs are put on a list and the list is passed to the prefetcher along with the usage profile that came in the HTTP header. After page display is complete, the prefetch algorithm runs, comparing the embedded hyperlinks with the usage profile. The comparison ensures that a recently deleted hyperlink mentioned in the usage profile will not be prefetched. The usage profile indicates the size of embedded HREFs (including the size of embedded images), and the prefetcher ensures that it never has GET requests outstanding for more than a certain fraction of the measured average bandwidth available to it: 50%, to be safe considering the inaccuracy of the bandwidth measure. Until the bandwidth limit is reached, the client prefetches URLs that have been accessed among the last 10 references, in descending order of popularity, down to a limit of 25%. That is, if the chances of a page being accessed are less than one quarter, that page is not prefetched. A prefetch request is not complete until the HTML page and all its embedded images have been loaded. A demand fetch aborts all prefetching efforts. As prefetch requests complete, more are issued from the "last-10" list and then from the "last-50" list, again in descending order of popularity down to the limit.

5 Implementation

There are two implementations of this approach to prefetching. In both, the server side is implemented by a proxy. The proxy emulates all the Web servers in the world, keeping track of usage reports and forming usage profiles for all URLs of all Web servers. This was done for testing and debugging purposes so that every Web server could seem to be one that supports prefetching. In practice, a server-side proxy might serve only a single server. The server-side proxy is an altered version of the W3C's `httpd`, version 3.0A.

The two implementations are distinguished by the client side. One implementation is an altered version of the September 4 1998 version of Mozilla which runs as multi-threaded software on UNIX. The other implementation is a proxy, once again an altered version of `httpd` 3.0A.

An important optimization is presently missing from the implementations: there is no need for a client to send a usage report to a server if both referer and referee URLs are on that server. The server (or its proxy) can determine the same information itself provided that the client sends the `Referer` field.

5.1 Evaluation

We have characterized prefetching performance through five measures: prefetch accuracy, client latency, network overhead, program space overhead, and program time overhead. All numbers are taken from the Mozilla implementation.

Mozilla has been altered to read the trace log and replay the HTML accesses with timing that is faithful (insofar as possible) to that in the log. The pages are displayed completely, just as if the user had typed in the same references with the timing evident in the log. The latency and prefetch accuracy numbers were gathered using this mechanism. Latency reduction is calculated by comparing the time between (1) Mozilla's initiation of an HTML page GET, and (2) Mozilla's receipt of the end of the last embedded image for that page. This time is compared to the similar time taken from the trace log. The two times

are not exactly comparable because the trace log records the time for a client-side snooping proxy to communicate with a remote web server, whereas Mozilla is recording the time for it to communicate with a server-side proxy which then communicates with the remote web server. However, since this approach places the prefetching implementation at a disadvantage – three parties communicate in series instead of two – we ignore the difference. Mozilla's disk cache size was kept at the default 5MB.

Prefetch Accuracy. The observed prefetch accuracy is high: while less than a majority of prefetched HTML pages (42.6%) are eventually accessed, a much higher fraction of all pages (62.5%) are eventually used. The reason is that many links from a common page share embedded images. So if page X has embedded links Y and Z, and if Y is wrongly prefetched instead of Z, considerable savings may still result if Y and Z share many embedded images. The overall increase in network traffic is considerably smaller (24.0%) than the overall prefetch miss rate of 37.5% because of demand fetches.

The restraint exercised by the prefetch algorithm – not prefetching links that have less than 25% chance of being accessed – governs the tradeoff between lowering latency and wasting bandwidth. Twenty-five percent was found to be the optimum point (from among every five percent) for balancing bandwidth waste against latency reduction.

Latency. Prefetching decreases average total latency to display an HTML page and all its embedded images by more than half (52.3%). This number probably lies between the prefetched-HTML hit rate (42.6%) and the overall prefetched hit rate (62.5%) because, on average, image pages are smaller than HTML pages.

Network Overhead. Usage reports and usage profiles can be lengthy. The average usage report is 66 bytes, while the average usage profile is 197 bytes. Most space is taken up by URLs, especially query URLs. It might be practical to abbreviate URLs in the usage profile, since the same URLs are embedded in the accompanying page content. However, no such method has been investigated.

Space Overhead. Some extra fields have been added to Mozilla's data structure that describes a page; however, this structure is very large and the added space is negligible. On the server side, the proxy's data structures grow in proportion to the number of pages distinct pages served.

Time Overhead. The effect of prefetch code on Mozilla's critical path is negligible, mostly because Mozilla executes a great deal of code for every GET operation. Time added at the server proxy is also negligible. The data structures for maintaining usage profiles are kept in memory, requiring added space but no extra disk accesses until they grow very large.

5.2 Implementation Effort

Mozilla was the largest and most difficult implementation, with a total of 3581 lines of code added or changed:

- Hooks into the parser to discover URLs: 47
- Response timing: 113
- Accumulating usage reports, tracking the state of negotiations with all servers: 786
- Prefetch algorithm: 1312
- Connection management: 751
- Cache management to address the Expires=0 problem: 450
- Short circuit of front-end display code so that pages are fetched but not displayed: 122

Three variations of the W3C `httpd` have been produced: a client-side prefetching proxy, a snooping proxy, and a server-side proxy that maintains usage statistics for all servers. The client-side proxy is the most complicated, though some code is shared with Mozilla; compared to Mozilla there are no front-end complications, and connection management is much easier. The snooping and server-side proxies are simple.

5.3 Privacy Implications

In order to operate transparently, the prefetching mechanism must examine HTML and `Referer` headers. This raises several privacy issues. The first is that the HTML must be available. End-to-end protocols or tunnels that might encrypt, compress, difference, or otherwise obscure HTML content could make the proxy implementation impossible. Second, some privacy advocates are concerned about the `Referer` field and want, at minimum, to be able to configure browsers not to send it. This goal is in conflict with our prefetching mechanism: suppressing the `Referer` field makes a proxy implementation impossible, while having a browser implementation send usage reports defeats `Referer` suppression. Indeed, a third privacy issue is that usage reports contain strictly more information than the `Referer` field. A fourth issue is that some users might feel uneasy knowing that the prefetcher examines their pages and browsing patterns. In this regard, the prefetcher does not pose a threat that is not already present from proxies, firewalls, and servers; nevertheless, knowing that the prefetcher systematically parses their pages might make some users uncomfortable. Finally, perhaps the largest privacy issue is that prefetching depends upon server administrators agreeing to release statistics about how their pages are being used. In many

cases such information has commercial value, meaning that web site operators might refuse to release it or demand payment for it.

6 Conclusion

We have shown that accurate Web prefetching is possible based on following HREFs in recently fetched pages. Letting the client control prefetching and aging of usage statistics has many advantages and may be the only practical approach in a world where proxies are commonplace. However, placing control at the client is also problematic because many pages contain large numbers of hyperlinks, and simply prefetching them all is worse than nothing. Also the client cannot be expected to have a good understanding of HREF reference patterns unless the page is one read frequently and/or the page rarely changes.

Our approach to this problem is to have clients pass record of their references up to the relevant server, which then distributes them to all clients. It is hypothesized that HREFs within a page are strongly skewed to "hot" and "cold," so that one client can learn from the usage patterns of others. The results in Section 5.1 bear this out.

The information exchange between clients and servers complicates deployment; however, there are other examples of recent work that depend on a similar flow of information [23, 7], suggesting that the idea may be useful more generally.

References

[1] V. Almeida et al. Characterizing Reference Locality in the WWW. In Proc. IEEE Conf. on Parallel and Distributed Information Systems. IEEE, December 1996. http://www.cs.bu.edu/~best/res/papers/pdis96.ps

[2] A. Bestavros. Using Speculation to Reduce Server Load and Service Time on the WWW. In *Proc. 4th ACM Intl. Conf. on Information and Knowledge Mgmt.*. ACM, November 1995. http://www.cs.bu.edu/~best/res/papers/cikm95.ps

[3] A. Bestavros and C. Cunha. Server-Initiated Document Dissemination for the WWW. *IEEE Data Engineering Bull.*, 19(3):3–11, September 1996. http://www.cs.bu.edu/~best/res/papers/debull96.ps

[4] R. Caceres, et al. Web Proxy Caching: The Devil is in the Details. *ACM SIGMETRICS Performance Evaluation Rev.*, 26(3):11–15, December 1998. http://www.research.att.com/~ramon/papers/wisp98.ps.gz

[5] K. Chinen and S. Yamaguchi. An Interactive Prefetching Proxy Server for Improvement of WWW Latency. In *Proc. INET 97*, June 1997. http://www.isoc.org/inet97/proceedings/A1/A1_3.HTM

[6] E. Cohen and H. Kaplan. Reducing User-Perceived Latency by Prefetching Connections and Pre-warming Servers *Unpublished AT&T technical report*, February 1999.

[7] E. Cohen, B. Krishnamurthy, and J. Rexford. Improving End-to-End Performance of the Web Using Server Volumes and Proxy Filters. In *Proc. ACM SIGCOMM 98*, pages 241–253. ACM, September 1998. http://www.research.att.com/~edith/Papers/sigcomm98.ps.Z

[8] Traces of Corporate Web Proxies. Compaq Corp. ftp://ftp.digital.com/pub/DEC/traces/proxy/webtraces.html

[9] M. Crovella and P. Barford. The Network Effects of Prefetching. In *Proc. Infocom 98*. IEEE, April 1998. http://cs-www.bu.edu/faculty/crovella/paper-archive/infocom98.ps

[10] M. Crovella and P. Barford. The Network Effects of Prefetching. Technical Report TR-97-002, Boston University, February 1997. http://cs-www.bu.edu/techreports/97-002-prefetcheff.ps.Z

[11] C.R. Cunha and C.F.B. Jaccoud. Determining WWW User's Next Access and Its Application to Pre-fetching. In *Proc. Second IEEE Intl. Symp. on Computers and Communication '97*. IEEE, July 1997. http://cs-www.bu.edu/techreports/97-004-userbehaviorprediction.ps.Z

[12] C.R. Cunha. Trace Analysis and Its Applications to Performance Enhancements of Distributed Information Systems. PhD Thesis TR-97-004, Boston University, 1997. http://www.cs.bu.edu/students/alumni/carro/thesis.ps.Z

[13] L. Fan et al. Web Prefetching Between Low-Bandwidth Clients and Proxies: Potential and Performance. In *Proc. ACM SIGMETRICS Conf.*, pages 178-187. ACM, May 1999. http://www.cs.wisc.edu/~cao/papers/prepush.ps.gz

[14] A. Feldmann et al. Performance of Web Proxy Caching in Heterogeneous Bandwidth Environments. In *Proc. Infocom 99*. IEEE, March 1999. http://www.research.att.com/~ramon/papers/infocom99.proxy.ps.gz

[15] Traces of Dialup Users. S. Gribble. http://ita.ee.lbl.gov/html/contrib/UCB.home-IP-HTTP.html

[16] Z. Jiang and L. Kleinrock. Prefetching Links on the WWW. In *ICC 97*. June 1997. http://millennium.cs.ucla.edu/~jiang/Research/Publication/prefetch.ps

[17] Z. Jiang and L. Kleinrock. An Adaptive Network Prefetch Scheme. *IEEE Journ. Selected Areas of Communication*, 17(4):358–368, April 1998. http://millennium.cs.ucla.edu/~jiang/Research/Publication/extended.ps

[18] M.F. Kaashoek, T. Pinckney, and J.A. Tauber. Dynamic Documents: Mobile Wireless Access to the WWW. In *Wkshp. on Mobile Computing Systems and Applications*, pages 179–184. IEEE, December 1994.

[19] T.M. Kroeger, D.D.E. Long, and J.C. Mogul. Exploring the Bounds of Web Latency Reduction from Caching and Prefetching. In *Proc. USENIX Symp. on Internet Technologies and Systems*, pages 13–22. USENIX, December 1997. http://www.usenix.org/publications/library/proceedings/usits97/kroeger.html

[20] T.S. Loon and V. Bharghavan. Alleviating the Latency and Bandwidth Problems in WWW Browsing. In *Proc. USENIX Symp. on Internet Technologies and Systems*, pages 219–230. USENIX, December 1997. http://www.usenix.org/publications/library/proceedings/usits97/tong.html

[21] E.P. Markatos and C.E. Chronaki. A Top-10 Approach to Prefetching on the Web. In *Proc. INET 98*. July 1998. http://www.ics.forth.gr/proj/arch-vlsi/html_papers/INET98_prefetch/paper.html

[22] J. Mogul et al. Potential Benefits of Delta Encoding and Data Compression for HTTP. In *Proc. ACM SIGCOMM 97*, pages 181–194. ACM, September 1997. ftp://ftp.digital.com/%7emogul/sigcomm97.ps.gz

[23] J. Mogul and P. Leach. Simple Hit-Metering and Usage-Limiting for HTTP. *IETF Network Working Group RFC 2227*, October 1997. http://www.ietf.org/rfc/rfc2227.txt

[24] NetAccelerator 2.0 software. http://www.imsisoft.com/netaccelerator/netacc2.html

[25] V. Padmanabhan and J. Mogul. Using Predictive Prefetching to Improve World Wide Web Latency. *Computer Communication Rev.*, 26(3):22–36, July 1996. http://www.cs.berkeley.edu/~padmanab/papers/ccr-july96.ps

[26] PeakJet2000 software. http://www.peak.com/peakjet2long.html

[27] Z. Wang and J. Crowcroft. Prefetching in World Wide Web. In *Proc. Globecom 96*. IEEE, December 1996. http://www.bell-labs.com/user/zhwang/papers/prefetch.ps.Z

[28] Z. Wang and J. Crowcroft. Prefetching in World Wide Web. In *Proc. Global Internet*, pages 28–32. IEEE, November 1996. http://www.cs.columbia.edu/~hgs/InternetTC/GlobalInternet96/Wang9611_Prefetching.ps.gz

Mining Longest Repeating Subsequences to Predict World Wide Web Surfing

James Pitkow
Xerox PARC
Palo Alto, CA 94304 USA
pitkow@parc.xerox.com

Peter Pirolli
Xerox PARC
Palo Alto, CA 94304
pirolli@parc.xerox.com

Abstract

Modeling and predicting user surfing paths involves tradeoffs between model complexity and predictive accuracy. In this paper we explore predictive modeling techniques that attempt to reduce model complexity while retaining predictive accuracy. We show that compared to various Markov models, longest repeating subsequence models are able to significantly reduce model size while retaining the ability to make accurate predictions. In addition, sharp increases in the overall predictive capabilities of these models are achievable by modest increases to the number of predictions made.

1. Introduction

Users surf the World Wide Web (WWW) by navigating along the hyperlinks that connect islands of content. If we could predict where surfers were going (that is, what they were seeking) we might be able to improve surfers' interactions with the WWW. Indeed, several research and industrial thrusts attempt to generate and utilize such predictions. These technologies include those for searching through WWW content, recommending related WWW pages, and reducing the time that users have to wait for WWW content downloads, as well as systems for analyzing the designs of web sites. Our previous work [12, 17] has attempted to characterize basic empirical properties of user surfing paths at web sites. Studied properties include the distribution of the number of clicks made by surfers at a web site, the complexity of path structures, and the consistency (or change) of paths over time. In this paper we explore pattern extraction and pattern matching techniques that predict future surfing paths. We expect that future work may exploit these modeling techniques in applications such as WWW search, recommendations, latency reduction, and analysis of web site designs.

Modeling and predicting user surfing paths involves tradeoffs between model complexity and predictive accuracy. In this paper we explore predictive modeling techniques that attempt to reduce model complexity while retaining predictive accuracy. The techniques merge two methods: a web-mining method that extracts significant surfing patterns by the identification of *longest repeating subsequences* (LRS) and a pattern-matching method that embodies the principle of *weighted specificity*. The LRS technique serves to reduce the complexity of the model by focusing on significant surfing patterns. This technique has been explored in connection with other areas of user interaction [8]. The weighted specificity principal exploits the finding that longer patterns of past surfing paths are more predictive. These techniques are motivated by results from previous predictive models and our prior empirical characterization [17] of surfing data. We shall show that when compared against a base standard of two different Markov model representations, the LRS pattern extraction and weighted specificity pattern matching techniques are able to dramatically reduce model complexity while retaining a high degree of predictive accuracy.

1.1. Surfing Paths

Figure 1 models the diffusion of surfers through a web site [12, 17]: (a) users begin surfing a web site starting from different entry pages, (b) as they surf the web site, users arrive at specific web pages having traveled different surfing paths, (c) users choose to traverse possible paths leading from pages they are currently visiting and (d) after surfing through some number of pages, users stop or go to another web site. Research in a number of application areas assumes, either explicitly or implicitly, that information about surfing paths observed in the past can provide useful information about surfing paths that will occur in the future.

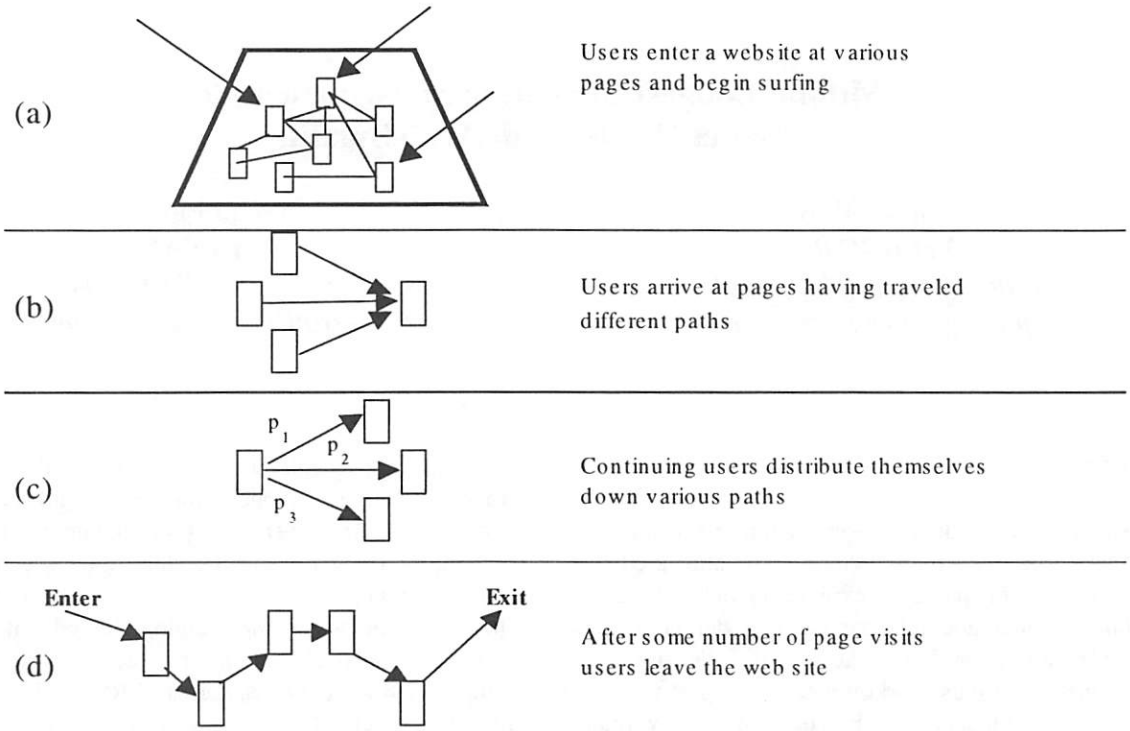


Figure 1. A conceptual model depicting the various stages of users traversing a web site.

2. Applications of Predictive Models

2.1 Search

The ability to accurately predict user surfing patterns could lead to a number of improvements in user-WWW interaction. For instance, the Google search engine [3] assumes that a model of surfing can lead to improvements in the precision of text-based search engine results. Conceptually, Google models surfers pursuing random walks over the entire WWW link structure. The distribution of visits over all WWW pages is obtained from this model. This distribution is then used to re-weight and re-rank the results of a text-based search engine. Under this model, surfer path information is viewed as an indicator of user interests, over and above the text keywords entered into a standard search engine. Following this line of reasoning one might also assume that surfing models with higher predictive accuracy would yield better search engines since the models provide a more realistic view of real world usage. The approach we propose here aims to be more informed than the random walk model implicit in Google. To the extent that surfing predictions improve text-based search results, we would expect that a more informed approach would yield better improvements than a random walk model.

2.2 Recommendation of Related Pages

Recently, tools have become available for suggesting related pages to surfers [10, 16, 18]. The "What's Related" tool button on the Netscape browser developed by Alexa, provides recommendations based on content, link structure, and usage patterns. Similar tools for specific repositories of WWW content are also provided by Autonomy. One can think of these tools as making the prediction that "surfers who come to this page (site) are most likely to be interested in the following pages (sites)." The predictive model of surfing proposed here could be used to enhance the recommendations made by these and other systems.

2.3 Web Site Models

Producers of WWW content are often interested in improvements in web site design. Recent research has developed visualizations to show the flow of users through a web site [5]. Businesses have emerged (e.g., Web Techniques, www.webtechniques.com) that send simulated users through existing web sites to provide data on web site design. Predictive models of surfer paths could help move the state of web site analysis from post-hoc modeling of past user interactions, or a current web site, to predictive models that can accurately simulate surfer paths through hypothetical

web site signs. Web site designers could explore different arrangements of links that promote desired flows of surfers through content.

2.4 Latency Reduction

Predictive models have significant potential to reduce user-perceived WWW latencies. Year after year, users report WWW delays as their number one problem in using the WWW [19]. One emerging strand of research that aims to improve WWW access times has grown out of research on improving file access time though prefetching and caching methods (e.g., [4]). Of particular interest to our research, Griffioen and Appleton's work [11] on file system prediction introduced the notion of automatic prefetching and evaluated the effectiveness of a one-hop Markov model.

A number of recent methods based on the use of Markov models as well as other methods have recently been proposed for prefetching and caching of WWW pages [1, 9, 13, 15, 20]. Roughly, the idea is that if a system could predict the content a surfer was going to visit next, and there was little cost involved, then the system could prefetch that content. While the user processes one page of content, other pages could be prefetched from high-latency remote sites into low-latency local storage.

Kroeger, Long, and Mogul [13] explored potential improvements in WWW interaction latencies that might be gained by predicting surfer paths. Current proxy servers typically mediate WWW access by accepting requests from user client applications. These requests are serviced by delivering content that has been cached, prefetched, and retrieved from other caches, or retrieved directly from origin Web servers. Proxies are typically accessed faster by clients than WWW servers are accessed by proxies. This usually occurs because the proxies are located on the same local area network as the client, whereas the proxies must access WWW servers over external network connections. Assuming this standard configuration, Kroeger et al. divided user-WWW latencies into (a) internal latencies caused by computers and networks utilized by the clients and proxies and (b) external latencies caused by computers and networks between the proxies and external WWW servers. Examining WWW user traces collected at the Digital Equipment Corporation WWW proxy, Kroeger et al. found that external latencies accounted for 88% of the total amount of latency seen by users geographically close to proxies. Other analyses by Kroeger et al. suggest that up to 60% of the observed external latencies could be reduced by improved caching and prefetching methods.

3. Predictive Surfing Models

Several predictive models of surfing have been developed in order to improve WWW latencies. It is instructive to review how their effectiveness varies. This review, plus a review of prior empirical characterizations of surfing patterns, motivate the pattern extraction and pattern matching techniques that we present in Section 5.1.

3.1 Path Profiles

Schechter, Krishnan, and Smith [20] utilized path and point profiles generated from the analysis of Web server logs to predict HTTP requests. They used these predictions to explore latency reductions through the pre-computation of dynamic Web pages. The profiles are constructed from user session. During a single session, a user interacting with the WWW traverses some sequence, S, of URLs. From that single sequence, the set of all possible subsequences is extracted as paths. Over some time period (say a day), the frequency of all observed paths is recorded. The resulting path profile consists of the set of all ordered pairs of paths and their observed frequencies.

Schechter et al. propose a method for predicting the next move of a surfer based on matching the surfer's current surfing sequence against the paths in the path profile. The ranking of matches is determined by a kind of specificity heuristic: the maximal prefixes of each path (the first N-1 elements of an N-length path) are compared element-wise against the same length suffixes of the user path (i.e., a size N-1 prefix is matched against the last N-1 elements of the user path), and the paths in the profile with the highest number of element-wise matches are returned. Partial matches are disallowed. In other words, if a surfer's path were <A, B, C>, indicating the user visited URL A, then URL B, then URL C, the path would be better matched by a path in the profile of <A, B, C, D> than <B, C, E>. For the paths in the profile that match, the one with the highest observed frequency is selected and used to make the prediction. Using our example, if <A, B, C, D> were the best match and most frequently observed path in the profile, then it would be used to predict that the user who just visited <A, B, C> would next visit URL D. Schechter et al. found that storing longer paths in the profile offered some improvements in prediction but they did not study this systematically.

Schechter et al. were also concerned with reducing model complexity. They pointed out that since a session S consists of an ordered sequence of URLs visited by a user, the worst case scenario of a naïve algorithm to store the decomposition of every path is order $|S|^3$. To reduce the model size, Schechter et al. used a maximal prefix trie with a minimal threshold requirement for repeating prefixes.

3.2 First-Order Markov Models

Based on the first-order Markov prediction method described in [11] for file prediction, Padmanabhan and Mogul [15] constructed a dependency graph containing nodes for all files ever accessed at a particular WWW server. Dependency arcs between nodes indicated that one file was accessed within some number of accesses w of another file. The arcs were weighted to reflect access rates. Padmanabhan and Mogul found that a predictive prefetching method based on this dependency representation reduced WWW latencies, and the reductions increased as w increased from w=2 to w=4.

Bestravos [1] used a method where one first estimates the conditional probabilities of transitioning directly from each page to every other page within a time T_w based upon server log file analysis. Like Padmanabhan and Mogul, this is a first-order Markov model approximation for predicting surfer paths, except the forward window is measured by time instead of number of pages. Using this approach, Bestravos developed a *speculative* service method that substantially reduced server loads and latencies. The architecture allows for the server and/or client to initiate the retrieval of resources predicted to be requested in the near future. Such systems now are generally referred to as *hint-based* systems, e.g., [7, 13, 15]. Bestravos did not, however, explore the effects of using longer surfer paths (higher-order Markov models) in the predictive model.

3.3 Summary

Regardless of where the transitions were recorded (proxy, server, etc.) all of these prefetching methods essentially record surfing transitions and use these data to predict future transitions. It is interesting to note that the methods of Schechter et al. and Padmanabhan and Mogul seemed to improve predictions when they stored longer path dependencies. In order to further motivate the rational behind the specificity principle for path matching, we next summarize the results of empirical analyses we [17] performed on server log files. In Section 5 we introduce the longest repeating subsequence (LRS) method as a technique that adheres to the principles of path specificity and model complexity reduction and evaluate it against surfing path data in Section 6.

4. Empirical Analysis Surfing Paths using Kth-Order Markov Models

In [17], we systematically evaluated the predictive capabilities of Kth-order Markov using ten days of log files collected at the xerox.com web site. Among other things, the results of this analysis suggest that storing longer path dependencies would lead to better prediction accuracy. This section reviews the methods and results of that study.

4.1 N-Gram Representation of Paths

Surfing paths can be represented as *n-grams*. N-grams can be formalized as tuples of the form $<X_1, X_2, \ldots X_n>$ to indicate sequences of page clicks by a population of users visiting a web site. Each of the components of the n-gram take on specific values $X_i = x_i$ for a specific surfing path taken by a specific user on a specific visit to a web site.

Users often surf over more than one page at a web site. One may record surfing n-grams, $<X_1, X_2, \ldots X_n>$ of any length observable in practice. Assume we define these n-grams as corresponding to individual surfing sessions by individual users. That is, each surfing session is comprised of a sequence of visits made by a surfer, with no significantly long pauses. Over the course of a data collection period—say a day—one finds that the lengths, n, of surfing paths will be distributed as an inverse Gaussian function[1], with the mode of the distribution being length one. This appears to be a universal law that is predicted from general assumptions about the foraging decisions made by individual surfers [12]. In practice one typically finds that the majority of users visit one page on a web site and then click to another web site.

4.2 Kth-Order Markov Approximations

The first-order Markov model used by Bestravos [1] and Padmanabhan and Mogul [15] were concerned with page-to-page transition probabilities. These can be

[1] The inverse Gaussian distribution is a heavily skewed distribution (much like the log normal distribution) that predicts that the bulk of recorded paths will be very short with a few very long paths.

estimated from n-grams of the form $<X_1, X_2>$ to yield the conditional probabilities

$$p(x_2 \mid x_1) = \Pr(X_2 = x_2 \mid X_1 = x_1).$$

If we want to capture longer surfing paths, we may wish to consider the conditional probability that a surfer transitions to an n^{th} page given their previous $k = n-1$ page visits:

$$p(x_n \mid x_{n-1},\ldots x_{n-k}) = \Pr(X_n = x_n \mid X_{n-1},\ldots, X_{n-k}).$$

Such conditional probabilities are known as K^{th}-order Markov approximations (or K^{th}-order Markov models). The zeroth order Markov model is the unconditional base rate probability:

$$p(x_n) = \Pr(X_n)$$

which is the probability of a page visit. This might be estimated as the proportion of visits to a page over a period of time.

4.3 Summary of Prior Empirical Analysis

Using data collected from xerox.com for the dates May 10, 1998 through May 19, 1998 we [17] systematically tested the predictive properties of K^{th}-order Markov models. The site received between 220,026 and 651,640 requests per day during this period. Over this period, there were 16,051 files on the xerox.com web site, of which 8,517 pages were HTML.

The reliable identification of user paths in a web site is often a complicated and site specific task. The xerox.com web site issued cookies to users only upon entry to the Xerox splash page and recorded the "User-Agent" and "Referer" field for each request when present. User paths were identified using cookies and a set of fallback heuristics when cookies did not exist[2]. The Xerox server permitted caching of resources. When present, Get-If-Modified headers were included in the construction of user paths. Still, under certain client and proxy configurations, the xerox.com caching policy resulted in missed client navigation, e.g., when a user clicked on the "Back" button. As a result, user paths constructed by our heuristics often contained transitions to other pages not linked to from the current page. Over the ten days used in this study, 176,712 user paths were observed.

[2] The exact methods, tradeoffs, and the effects of counting each IP as a user are described greater detail in [17].

The models were estimated from surfing transitions extracted from training sets of WWW server log file data and tested against test sets of data that occurred after the training set. The prediction scenario assumed a surfer was just observed making k page visits. In order to make a prediction of the next page visit the model must have (a) an estimate of $p(x_n|x_{n-1},\ldots x_{n-k})$ from the training data, which required that (b) a path of k visits $<x_{n-1},\ldots x_{n-k}>$ (a penultimate path) had been observed in the training data. Given a penultimate path match between paths in the training and test data, the model examined all the conditional probabilities $p(x_n|x_{n-1},\ldots x_{n-k})$ available for all pages x_n, and predicted that the page having the highest conditional probability of occurring next would in fact be requested next. Whether the observed surfer made the predicted visit (a hit) or not (a miss) was then tallied. The model did not make a prediction when a matching path in the model did not exist. It is important to examine the performance of the model in making correct predictions as well as incorrect predictions since incorrect predictions often result in undesirable costs, which can mitigate any benefits.

Table 1 presents a subset of the analyses presented in [17], where predictions based on a training set collected one day and were tested against data collected the next day. We define

- Pr(Match), the probability that a penultimate path, $<x_{n-1},\ldots x_{n-k}>$, observed in the test data was matched by the same penultimate path in the training data,
- Pr(Hit|Match), the conditional probability that page x_n is visited, given that $<x_{n-1},\ldots x_{n-k}>$, is the penultimate path and the highest probability conditional on that path is $p(x_n|x_{n-1},\ldots x_{n-k})$,
- Pr(Hit) = Pr(Hit|Match)•Pr(Match), the probability that the page visited in the test set is the one estimated from the training as the most likely to occur (in accordance with the method in our scenario),
- Pr(Miss|Match), the conditional probability that page x_n is *not* visited, given that $<x_{n-1},\ldots x_{n-k}>$, is the penultimate path and the highest probability conditional on that path is $p(x_n|x_{n-1},\ldots x_{n-k})$,
- Pr(Miss) = Pr(Miss|Match)•Pr(Match), the probability that the page visited in the test set is *not* the one estimated from the training as the most likely to occur (in accordance with the method in our scenario), and

Table 1. Probability of matching a path of the same length, Pr(Match), conditional probability of accurately predicting the next page visit of a surfer given a path match, Pr(Hit|Match), and the overall accuracy of predicting a surfer page visit, Pr(Hit). Conditional miss probabilities, Pr(Miss|Match), overall miss rate, Pr(Miss), and hit to miss ratios, Pr (Hit)/Pr(Miss), are also provided. Training data and test data were collected on successive days. Matching and predictions were conducted on paths of the same length. No predictions were made in the absence of a match.

Order of Markov Model	Pr (Match)	Pr (Hit|Match)	Pr (Hit)	Pr (Miss|Match)	Pr (Miss)	Pr(Hit)/ Pr(Miss)
1	.87	.23	.20	.77	.67	.30
2	.61	.30	.18	.70	.43	.42
3	.30	.34	.10	.66	.20	.51
4	.21	.41	.08	.59	.12	.65
5	.21	.29	.06	.71	.15	.40
6	.20	.31	.06	.69	.14	.43
7	.20	.27	.05	.73	.15	.34

- Pr(Hit)/Pr(Miss), the probability of correctly predicting page x divided by the probability of making an incorrect prediction for all transitions.

The last metric provides a coarse measure of the benefit-cost ratio. That is,

$$\text{Benefit:Cost} = B * \Pr(\text{Hit}) / C * \Pr(\text{miss})$$

where B and C vary between 0 and 1 and represent the relative weights associated with the benefits and costs for each application. Naturally, different applications have different inherent tradeoffs associated with the benefits of making a correct prediction versus the costs of making an incorrect prediction and will often require a more complex metric to encode the true tradeoffs. From Table 1 it can be seen that,

- Lower-order models have higher Pr(Match). This indicates that the chances of seeing short surfing paths across days are much higher than for longer surfing paths.
- Higher-order models have higher Pr(Hit|Match). If one can find a match of longer surfing paths then they are likely to be better predictors than shorter surfing paths.
- Lower-order models have higher Pr(Hit) overall. This indicates that the overall hit rate is dominated by the probability of finding a penultimate path match, Pr(Match).
- For the xerox.com data, if one assumes that the benefit of making a correct hit equals the cost of making an incorrect prediction ($B = C = 1$), using a 4^{th} order model is optimal.

Included in [17] is an analysis of prediction stability over time using entropy analysis and the improvements due to increasing the size of the training data set.

The results of the above analyses lead us to explore methods that would improve both pattern extraction and pattern matching. In this next section, we introduce the notion of longest repeating subsequences (LRS) to identify information rich patterns and present two modifications of LRS to improve pattern matching. These models are then compared to different Markov models in Section 6.

5. Model and Prediction Methods

Schechter et al. [20] discuss the explosive growth of storage requirements for path profiles (we discuss their analysis in detail below). Producing an accurate predictive model using the least amount of space has many computational benefits as well as practical benefits. One might even imagine a model being compact enough to reside in memory for each thread handling requests on a busy WWW server.

One solution to reduce model space is to use compact data structures like tries as was used in Schechter et al. While an issue of great interest and complexity, we will not treat more efficient data structure solutions in this paper. Another solution is to attack the problem at the core and remove low information elements from the model. The LRS technique treats the problem as a data-mining task, where some of the paths are considered noise. This hinges off the insight that many paths occur infrequently, often as a result of erroneous navigation. Identifying repeating subsequences enables common sub-paths to be extracted. This has the benefit of preserving the sequential nature of the paths while being robust to noise. Using LRS, the storage requirement is reduced by saving only information rich paths.

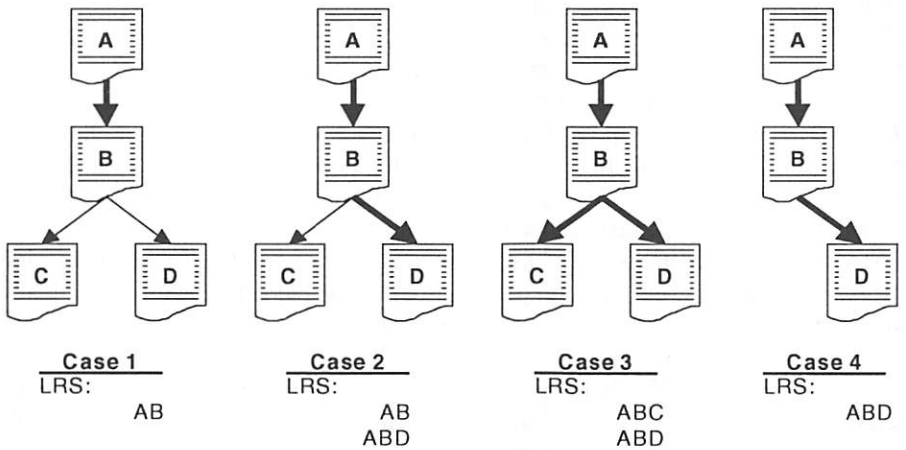


Figure 2. Examples illustrating the formation of longest repeating subsequences (LRS). Thick-lined arrows indicate more than one traversal whereas thin-lined arrows indicate only one traversal. For each case, the resulting LRS are listed.

The second method we explore is to use a specificity heuristic in pattern matching. As shown by the results of Schechter et al., Padmanabhan and Mogul, and Table 1, higher-order Markov models result in higher prediction rates when there is a penultimate path match. The principle of specificity encourages the use of higher-order path matches whenever possible to maximize hit rates. The drawback of this approach is that the likelihood of a higher-order path match is quite small, resulting in lower overall hit rates. The decreased likelihood of a long path match is in part a function of the inverse Gaussian distribution of path lengths, where over 80% of the distribution is accounted for by paths of length four or less. It also is a function of the exponential growth in the combinations of possible paths, where the length of the path is the base and the exponent is the average number of links per page within the web site. As an example, given a site with an average of three links per page, a path of length eight will have 6,561 possible combinations assuming only forward traversals. Due to limitations in the ability of server logs to completely capture user navigation, recorded paths are not limited to only forward links, which makes the set of possible combinations even larger.

5.1 Longest Repeating Sequences

A longest repeating subsequence [8] is a sequence of items where

1) subsequence means a set of consecutive items,
2) repeated means the item occurs more than some threshold T, where T typically equals one, and
3) longest means that although a subsequence may be part of another repeated subsequence, there is at least once occurrence of this subsequence where this is the longest repeating.

To help illustrate, suppose we have the case where a web site contains the pages A, B, C, and D, where A contains a hyperlink to B and B contains hyperlinks to both C and D. As shown in Figure 2, if users repeatedly navigate from A to B, but only one user clicks through to C and only one user clicks through to D (as in Case 1), the longest repeating subsequence is AB. If however more than one user clicks through from B to D (as in Case 2), then both AB and ABD are longest repeating subsequences. In this event, AB is a LRS since on at least one other occasion, AB was not followed by ABD. In Case 3, both ABC and ABD are LRS since both occur more than once and are the longest subsequences. Note that AB is not a LRS since it is never the longest repeating subsequence as in Case 4 for the LRS ABD.

LRS have several interesting properties. First, the complexity of the resulting n-grams is reduced as the low probability transitions are automatically excluded from further analysis. This reduction happens for all transitions that occur only T times, which in some cases, will result in a prediction not being made. For example, with a threshold of T=1 the penultimate match for the LRS AB is A in Case 1. In this case, a prediction will not be made after the pages A and B have been requested. The reduction in complexity typically results in a slight loss of pattern matching, as will be made evident in the below experiments.

To contrast, if all 2^{nd}-order Markov approximations were being used to make predictions and path AB has been observed, the system would have to randomly

select either C or D since each are equally probable. If a hint-based prefetching system were used, a set of predictions could also be made by the server and passed along to the client with the contents of the currently requested page. In this case, a list could be returned that contains both C and D, each with equal probability. We will study the effect of varying result list size in section 6.3. Note also that in Case 2, after seeing AB the LRS predictive model will predict D, just as would all the 2nd-order Markov approximations. In this manner, the LRS model can be viewed as a proper subset of the Kth-order Markov approximations.

Another interesting property of the LRS model is its bias towards specificity. Any single page-to-page transition that is always repeated as part of longer sequences is not included into the LRS model. For web sites, a transition from the popular entry points usually results in all forward links being followed more than some threshold T. For predictive purposes, this reduces the number of penultimate matches that the LRS model can make for shorter paths and as direct result, lowers the overall hit rate for the model.

5.2 Hybrid LRS-Markov Models

We now propose two straightforward hybrid models that use LRS subsequences extracted from training data. In the first hybrid LRS model we extract LRS patterns from the training data and use these to estimate a first-order Markov model. That is, we decompose each LRS pattern into a series of corresponding one-hop n-grams, e.g., the LRS ABCD would result in AB, BC, and CD 1-grams. We call this model a *one-hop LRS* model. In Section 6.1, we compare the one-hop LRS model against a first-order Markov model estimated from all the paths in a training data set.

The second hybrid LRS model decomposes the extracted LRS subsequences into all possible n-grams. The resulting model is a reduced set of n-grams of various lengths. We call this the *All-Kth-Order LRS* model, since all orders of k are able to make predictions. The main advantage of this model is that it incorporates the specificity principle of pattern matching by utilizing the increased predictive power contained in longer paths. Below, we test the All-Kth-Order LRS model against an *All-Kth-Order* Markov model. As with the All-Kth-Order LRS model the All-Kth-Order Markov model is constructed by decomposing all possible subsequences into varying length n-grams.

5.3 Model complexity reduction

Schechter et al. [20] note that their profiling method requires that a path of length S be decomposed into all subpaths. They point out that to store the profile for a single path of length S (i.e., all the necessary subpaths) requires storage that grows as $O(S^3)$. Note, however, that n distinct paths of length S will not necessarily each require $O(S^3)$ storage. The distinct paths of length S will share many redundant subpaths of length less than S. The problem with the Schechter et al. analysis is that it fails to incorporate the combinatorics of surfing paths. Moreover, it is only in the worst case that one needs to store all distinct paths and all distinct subpaths. One may have the goal of storing those that are likely to be needed, and this is the aim of the LRS pattern extraction method.

The amount of space required for all models—LRS and Markov—depends on the combinatorics (e.g., the redundancy) of user paths. Basically, we need to know how the *number of patterns* of paths and subpaths grows as a process of surfing. A model of the path combinatorics could be formulated if we understood the underlying path-generating process [6]. Unfortunately, the process that generates user paths within web sites has not been characterized in detail (although see [12] for beginnings) and may well very from site to site as well as from time to time.

In the absence of a more informed surfing process model, we explore the following simple model. Assume that surfing paths have an average branching factor b. That is, surfers may start in b places, and from each page they move to one of b pages on average. Assume that the surfers move at random, thereby generating random paths. Surfing paths of length S can be divided into S/k subpath partitions of length $0 < k \leq S$. Each subpath partition of length k will have b^k patterns (assuming randomly chosen paths along the b branches). So the complexity cost, $C(k)$, in terms of number of patterns as a function of subpath length k will be

$$C(S) = \sum_{i=1}^{S} (S/i) b^i$$

As noted by Newell and Rosenbloom [14], this does not have a closed-form solution but one can show that the derivative is such that

$$C'(S) = e^{\log(b)S}.$$

It should also be noted, that under this random process model longer path patterns are less likely to recur than shorter path patterns.

For the one-hop models, the worst case complexity of a one-hop Markov model is $O(V^2)$, where V is the number of pages in the web site and every page is connected to every other page. In practice, the connectivity of web pages is sparser as evidenced by the low number of hyperlinks per page [2]. Such sparse connectivity graphs are well suited for adjacency-list representations, which require $O(V+E)$ where E is the number of edges between pages V. As mentioned previously, the worst-case storage requirements for All-K^{th}-order Markov approximations are $O(S^3)$.

The result of applying LRS to a set of path data is a possible pruning of the resulting Markov models. The extent of the pruning depends on the amount of redundancy and the value selected for the repeating threshold T. In the worst case, the amount of storage required for the LRS models is equal to the worst-case for the equivalent Markov models. As we shall see, in practice, the reduction can be quite significant, though we note that the amount of reduction will likely vary from web site to web site and even within sites depending on traffic patterns.

6. Evaluation

In order to test whether the hybrid LRS models help achieve the goal of reducing complexity while maintaining predictive power, we conducted a set of experiments on the same data used in our previous study [17] summarized above. For the below analyses, three consecutive weekdays were chosen starting on Monday May 11, 1998 for the training data and Thursday May 14, 1998 for the test day. There were 63,329 user paths for the training days and 19,069 paths for the test day. As with our previous evaluation, predictions were not made when the model did not contain a matching pattern.

6.1 One-hop Markov and LRS Comparison

One-hop Markov models and their derivatives are important to study empirically as they are conceptually simple and easy to implement. Developing a rich understanding of these models helps frame the results and tradeoffs of more complex models. For this experiment, the one-hop Markov and the one-hop LRS models were built using three training days. For each path in the test day, the path was decomposed into a set of sequential transitions. Each model was then tested to see if there was a matching prefix (Match) for each path, and if so, if the correct transition was predicted (Hit). From this, the probability of a match Pr(Match), the probability of a hit given a match Pr(Hit|Match), the hit rate across all transitions Pr(Hit), and the benefit-cost ratio Pr(Hit)/Pr(Miss) were computed along with the corresponding miss probabilities.

Table 2 displays the results for the one-hop Markov model and the one-hop LRS model. Overall, there were 13,189 unique hops in the training data with the one-hop LRS model reducing this set by nearly a third to 4,953 unique hops. The hit probabilities in Table 2 are slightly higher than in Table 1. These discrepancies result from weekday and weekend traffic differing significantly at xerox.com and that the data used in Table 2 contained only weekday data for the training and testing sets.

Table 2. Results of comparing a one-hop Markov model with the one-hop LRS model.

	One-hop Markov	One-hop LRS
Number of one-hops	13,189	4,953
Model Size (bytes)	372,218	136,177
Total transitions in test data	25,485	25,485
Matches	25,263	24,363
Hits	6,402	6,056
Pr(Match)	.99	.96
Pr(Hit|Match)	.25	.25
Pr(Hit)	.25	.24
Pr(Miss|Match)	.75	.75
Pr(Miss)	.74	.72
Pr(Hit)/Pr(Miss)	.34	.33

The one-hop LRS model produces a reduction in the total size required to store the model thus satisfying the complexity reduction principle. One might expect that the sharp reduction in the model's complexity would result in an equally sharp reduction in predictive ability. However, this is not the case as the one-hop LRS model performs nearly as well as the one-hop Markov model in terms of the total number of predictions made (25,263 one-hop Markov versus 24,363 one-hop LRS), total number of hits (6,402 one-hop Markov versus 6,056 one-hop LRS), and the overall probability of correctly predicting the next page Pr(Hit) (25% one-hop Markov versus 24% one-hop LRS). Both models produced similar miss probabilities, with the Markov model resulting in a 75% incorrect prediction rate and the LRS model 72%. The benefit-to-cost ratio for each model is nearly identical.

The one-hop LRS model was able to significantly reduce model complexity while preserving predictive ability. However both these models are very simple and they do not leverage the greater predictive abilities of longer paths.

6.2 All-Kth-order Markov approximation and All-Kth-order LRS Comparison

In order to adhere to the principle of specificity, longer paths should be used wherever possible given their greater predictive power. With this in mind, we set out to explore the differences between the All-Kth-order Markov model and the All-Kth -order LRS model. As with the one-hop analysis above, since the All-Kth-LRS is a subset of all All-Kth-order Markov approximations, we did not expect to see better predictive capabilities, but rather wanted to examine the tradeoffs between complexity reduction and the model's predictive power.

Table 3. Results from testing the All-Kth-order Markov and the All-Kth-order LRS models.

	All-Kth-order Markov	All-Kth-order LRS
Number of one-hops	217,064	18,549
Model Size (bytes)	8,847,169	616,790
Total transitions in test data	25,485	25,485
Matches	25,263	24,363
Hits	7,704	6,991
Pr(Match)	.99	.99
Pr(Hit|Match)	.31	.27
Pr(Hit)	.30	.27
Pr(Miss|Match)	.69	.73
Pr(Miss)	.69	.72
Pr(Hit)/Pr(Miss)	.44	.38

For this experiment, the same three training days and test day were used. For each training day, each path was decomposed into all corresponding n-grams for the All-Kth-order Markov model. With the All-Kth-order LRS model, the LRS were first computed and then each LRS was decomposed into the set of all n-gram subsequences. For the evaluation, each transition in a path of the test data was broken down into its corresponding subsequence suffixes. The resulting suffixes were then checked to see if there was a matching n-gram for each model. The prediction with the greatest specificity (number of element-wise matches between training and test paths) weighted by conditional probability was then selected as the prediction. This weighted specificity measure differs slightly from that used in our previous study and that used by Schechter et al. A prediction was not made if a suffix match was not found.

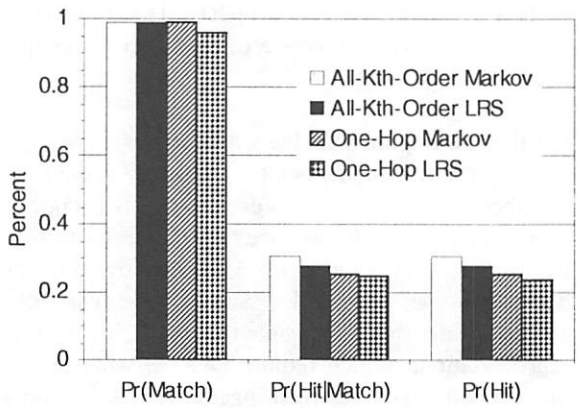


Figure 3. Summary of the likelihood of being able to make a prediction Pr(Match), correct prediction given a match pr(Hit|Match) and overall hit rate Pr(Hit) of each model.

Table 3 shows the results of the experiment. As with the previous experiments, the test data consisted of 25,485 transitions. As one would expect, the complexity of storing the All-Kth-order Markov is significant as there are close to one quarter million n-gram sequences which in a naive representation require 8,800 Kbytes. The All-Kth-order LRS model reduces the model space by an order of magnitude, decreasing the total number of n-grams to 18,549, which consume 616 Kbytes of space, which is nearly double that for the one-hop Markov model and almost six times that for the one-hop LRS model. The All-Kth-order Markov model was able to match nearly all transitions and resulted in a correct prediction 31% of the time. The performance of this model is better than the one-hop Markov model tested earlier, highlighting the specificity principle. The All-Kth-order LRS model performed nearly as well, correctly predicting the next page request 27% of the time, with an overall miss rate of 73% compared to 69% for the Markov model. The benefit-to-cost ratio for the Markov model exceeds that for the LRS model (.44 versus .38).

Figure 3 summarizes the results of the two experiments with respect to hit rates. In certain cases, the difference between the predictive power of the All-Kth may not outweigh the considerable space savings of the one-hop models. While the All-Kth-order Markov model provides the highest hit rate, the one-hop Markov model provides 83% of the predictive power while consuming only 4.2% of the space. The same is true for the one-hop LRS, where 80% of the predictive power is accomplished using only 1.5% of the space.

6.3 Parameterization of Prediction Set

Restricting the prediction to only one guess imposes a rather stringent constraint. One could also predict a larger set of pages that could be surfed to on the next click by a user. For certain applications that aim to reduce user latency like hint-based perfecting architectures, it is important to understand the cost-benefit tradeoffs in a systematic manner. This section explores how the hit rate varies when considering larger sets of predictions.

We evaluated each model's performance when returning sets of between one and ten predictions. Each set was constructed by ranking the predictions about the next page visit in decreasing order of likelihood, and selecting the topmost n predictions. For the one-hop models, the predictions were ranked by probability of predicted transition. For the All-Kth–order models, the predictions were ranked by the weighted specificity principle.

Figure 4 shows the probability of each model correctly predicting the next page visit across different prediction set sizes. As with the previous experiments, the All-Kth-order models performed better than the one-hop models due to the increased information contained in the longer paths and the LRS models performed slightly worse than the Markov models at a fraction of the model space. Increasing the prediction set has a dramatic impact on predictive power, with the predictive power of each method nearly doubling by increasing the set size to four elements.

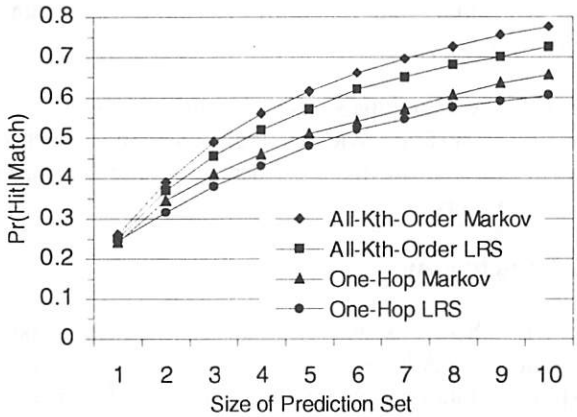


Figure 4. The effect on making a correct prediction as a function of the number of predictions made by each model.

7. Future Work

Motivated by the principle of weighted path specificity and complexity reduction, we have shown that small compact models of user surfing behavior exist that retain predictive power. Clearly, other methods exist to predict future access of resources. While this paper has focused on enhancements to various Markov models, we believe that the concept of LRS can be successfully applied to Markov models in other domains as well as to other suffix-based methods.

In the above experiments, subsequences that occurred more than once were considered repeating. In theory, repeating can be defined to be any occurrence threshold T. We hypothesize that model complexity can be reduced even further while maintaining acceptable predictive power by raising the threshold T. Determining the ideal threshold will depend upon the specific data and the intended application.

Another variable that was not parameterized in the above experiments was the confidence level for each prediction. Inspection of the data revealed that in several cases, each model was making predictions that were not very likely. A modified pattern-matching algorithm could be restricted to only make predictions when a given probability of making a successful prediction was achieved. While lowering the probability of a match Pr(Match), the reduced overall hit Pr(Hit) rate could be offset by the increased likelihood of being correct Pr(Hit|Match). This is especially appropriate for applications where the cost of being wrong outweighs the benefits of being right.

The application of the LRS models to prefetching and latency reduction is also of interest. Given the compact size of the LRS models, one can imagine HTTP server threads issuing hint lists to clients while maintaining the model in memory. Our preference is a system in which the server provides the client with a list of suggested prefetchable items. The client then decides what items to prefetch when. This would not require, but might benefit greatly from modifying the current pre-computed static LRS model into an adaptive, real-time model, especially since the optimal hint set size will most likely vary from server to server as well as page to page within a server. The overall effectiveness of this application and modifications needs to be evaluated. In a similar manner, LRS provides a compact information-dense method to store user paths for later data analysis.

From a more psychological perspective, we note that LRS may represent the common navigational sub-units or "chunks" across all users and documents on a web site. That is, the repeating subsequences may be an appropriate logical unit to encode the paths most traveled. We postulate that these chunks are well suited

for document and user clustering since they preserve the sequential nature of surfing, are robust against noise, and reduce overall computational complexity.

Finally, the exact space reduction achievable by LRS for Web surfing requires the generating function underlying web surfing to be identified and other traces to be examined.

8. Conclusion

Clearly there exists a tradeoff between model complexity and predictive power. Our initial exploration into the predictive capabilities of user paths led us to postulate the principles of complexity reduction and specificity. From this, we employed the notion of longest repeating subsequences to produce a subset of all paths. We showed that in the simplest case of modeling paths as a one-hop Markov model, the reduced one-hop LRS model was able to match the performance accuracy of the one-hop Markov model while reducing the complexity by nearly a third. We then showed that overall hit rates could be raised by including the principle of specificity, with the All-K^{th}-Order LRS model almost equaling the performance of the All-K^{th}-order Markov model while reducing the complexity by over an order of magnitude. We further showed that varying the size of prediction set results large gains in predictive.

9. Acknowledgements

We would like to the USENIX reviewers and our shepherd Jeff Mogul for their helpful comments and suggestions.

10. References

1. Bestravos, A. (1995). Using speculation to reduce server load and service time on the WWW. *Proceedings of the 4th ACM International Conference on Information and Knowledge Management, (CIKM '95)* Baltimore, MD.

2. Bray, T. (1996). Measuring the Web. *Proceedings of the Fifth International WWW Conference* Paris, France.

3. Brin, S. and Page, L. (1998). The anatomy of a large-scale hypertextual web search engine. *Proceedings of the Seventh International WWW Conference* Brisbane, Australia.

4. Cao, P., Felten, E.W., Karlin, A.R., and Li, K. (1996). Implementation and performance of integrated application-controlled file caching, prefetching, and disk scheduling. *ACM Transactions on Computer Systems, 14*, 311-343.

5. Chi, E., Pitkow, J., Mackinlay, J., Pirolli, P., Gossweiler, R., and Card, S.K. (1998). Visualizing the evolution of web ecologies. *Proceedings of the Conference on Human Factors in Computing Systems, (CHI '98)* Los Angeles, CA.

6. Clement, J., Flajolet, P., and Valle, B. (1998). The analysis of hybrid trie structures. *Proceedings of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms.*

7. Cohen, E., Krishnamurthy, B., and Rexford, J. (1998). Improved end-to-end performance of the web using server volumes and proxy filters. *Proceedings of the ACM SIGCOM.*

8. Crow, D. and Smith, B. (1992). DB_Habits: Comparing minimal knowledge and knowledge-based approaches to pattern recognition in the domain of user-computer interactions. In R. Beale and J. Finlay (Eds.), *Neural networks and pattern recognition in human-computer interaction* (pp. 39-63). New York: Ellis Horwood.

9. Cunha, C.R. (1997). *Trace analysis and its applications to performance enhancements of distributed information systems*. Unpublished thesis, Boston University, Boston.

10. Dean, J. and Henzinger, M.R. (1999). Finding related pages in the World Wide Web. *Proceedings of the Eighth International World Wide Web Conference* Toronto, Canada.

11. Griffioen, J. and Appleton, R. (1994). Reducing file system latency using a predictive approach. *Proceedings of the 1994 Summer USENIX Technical Conference* Cambridge, MA.

12. Huberman, B.A., Pirolli, P., Pitkow, J., and Lukose, R.J. (1998). Strong regularities in World Wide Web surfing. *Science, 280*, 95-97.

13. Kroeger, T.M., Long, D.D.E., and Mogul, J.C. (1997). Exploring the bounds of web latency reduction from caching and prefetching. *Proceedings of the USENIX Symposium on Internet Technologies and Systems, (USITS '97)* Monterey, CA.

14. Newell, A. and Rosenbloom, P.S. (1981). Mechanisms of skill acquisition and the law of practice. In J.R. Anderson (Ed.) *Cognitive skills and their acquisition* (pp. 1-55). Hillsdale, NJ: Lawrence Erlbaum.

15. Padmanabhan, V.N. and Mogul, J.C. (1996). Using predictive prefetching to improve World Wide Web latency. *Computer Communication Review, 26*, 22-36.

16. Pirolli, P., Pitkow, J., and Rao, R. (1996). Silk from a sow's ear: Extracting usable structures from the web. *Proceedings of the Conference on Human Factors in Computing Systems, (CHI '96)* Vancouver, Canada.

17. Pirolli, P. and Pitkow, J.E. (1999). Distributions of surfers' paths through the World Wide Web: Empirical characterization. *World Wide Web, 2(1-2)*, 29-45.

18. Pitkow, J. and Pirolli, P. (1997). Life, death, and lawfulness on the electronic frontier. *Proceedings of the Conference on Human Factors in Computing Systems, (CHI '97)* Atlanta, GA.

19. Pitkow, J.E. and Kehoe, C.M. (1999). GVU's Tenth WWW User Survey. *Online Publication: http://www.gvu.gatech.edu/user_surveys.*

20. Schechter, S., Krishnan, M., and Smith, M.D. (1998). Using path profiles to predict HTTP requests. *Proceedings of the Seventh International World Wide Web Conference* Brisbane, Australia.

Active Names: Flexible Location and Transport of Wide-Area Resources*

Amin Vahdat
Department of Computer Science
Duke University

Michael Dahlin
Department of Computer Science
University of Texas, Austin

Thomas Anderson
Amit Aggarwal
Department of Computer Science and Engineering
University of Washington

Abstract

In this paper, we explore flexible name resolution as a way of supporting extensibility for wide-area distributed services. Our approach, called Active Names, maps names to a chain of mobile programs that can customize how a service is located and how its results are transformed and transported back to the client. To illustrate the properties of our system, we implemented prototypes of server selection based on end-to-end performance measurements, location-independent data transformation, and caching of composable active objects and demonstrate up to a five-fold performance improvement to end users relative to protocols in widespread use. We show how these new services are developed, composed, and secured in our framework. Finally, we develop a set of algorithms to control how mobile Active Name programs are mapped onto available wide-area resources to optimize performance and availability.

1 Introduction

In this paper, we address the question: what should the architecture be for deploying advanced distributed services across the Internet? We argue for a programmable naming abstraction called Active Names that combines location and transport of resources, provides end-to-end programmability, supports composibility of different extensions, and supports mobile location-independent code.

This approach is motivated both by efforts to extend the current Internet Domain Naming System, DNS [39], and by other recent efforts to interpose services between clients and servers. Historically, DNS was developed to support a simple one-to-one mapping from machine names to IP addresses, but today Internet services identified by a single name are often distributed across many machines. This has lead researchers to enhance applications [8], routers [15], and DNS [34] to support more complex mappings. More broadly, academic and industrial researchers have proposed a bewildering array of new services for mediating between clients and servers, including dynamic redistribution of replicas over the wide area [50], compression and distillation of multimedia content [4, 23], proxy cache extensions such as support for hit counting and dynamic content [12, 49], client customization of web content [55], network address translators (NATs) [21], packet delivery services for mobile hosts [14], and so forth.

One approach to deploying such extensions would be to develop a new protocol that closely coordinates browsers, proxies, servers, and the name system to support these extensions. But such a system would be complex and difficult to modify, given the ongoing demand for improvements to existing services and the development of new services. To address this need for rapid deployment and extensibility of Internet services, a variety of proposals have been made to support "active" (dynamically migratable) computation inside the network, leveraging machine-independent languages such as Java [24]. Example "active" networking proposals include Active Networks [54], Active Caches [12], Active Services [4], RentAServer [50], and others. In addition, a number of proposals have been made for single point (non-migratable) extension code, such as HTTP front ends [23], NATs, and router-based load balancing [15]. These approaches have varied in where in the protocol stack the computation is applied – from packet filtering to connection filtering to transparent proxies to giving up on transparency and making the application responsible for everything.

If active computation can be applied anywhere in the protocol stack, where in the protocol stack *should* it be done? In this paper, we argue for *active naming* as a

*This work was supported in part by the Defense Advanced Research Projects Agency (F30602-95-C-0014, F30602-98-1-0205), the National Science Foundation (CDA 9401156, CDA 9624082), Cisco Systems, Dell, Sun Microsystems, Novell, Hewlett Packard, Intel, Microsoft, and Mitsubishi. Anderson was also supported by a National Science Foundation Presidential Faculty Fellowship. Dahlin was also supported by an NSF CAREER grant (CCR 9733742).

unifying principle to efficiently support the composibility of a wide variety of new services while providing correct end-to-end behavior. Our Active Names system maps a name to a chain of mobile programs responsible for locating the remote service and transporting its response to the destination. A service owning a portion of the namespace has complete control over which protocols are used to access the service along with control over where in the network those protocols run. Similarly, a client machine can use Active Names to customize how services are presented to the user. For example, when a mobile user with a small screen and expensive wireless connection in Europe refers to "cnn.com", the user probably wants to go to a different replica, fetch different data, and transform that data differently than a user in the United States with a large screen and T3 connection.

To illustrate our approach, we have constructed a prototype Active Naming system and have used it to implement and study a set of complex distributed services including (i) replica selection based on end-to-end performance observations, (ii) image retrieval protocols that migrate distillation code dynamically around the network to optimize client response time, and (iii) a cache enhanced to support both client customization and caching of active content. These experiments demonstrate up to five-fold performance improvements for end-users of our system relative to protocols in widespread use today.

Of course, no performance improvement can "prove" that extensibility is a good idea; any algorithm that demonstrates large gains in an extensible system could, in principle, be deployed as part of a new, non-extensible protocol. Instead of trying to resolve the argument of whether extensibility is desirable, we begin with the premise that it is, and then focus on how best to support advanced, extensible and high performance Internet services. First, our experiments show that our Active Names system is "complete" in that a wide variety of extensions can be easily implemented in our system, and that these extensions can significantly improve performance compared to existing protocols. Second, our experiments suggest that high performance demands a combination of three features found together in Active Names but only individually in other systems: location-independent program execution, efficient composibility of extensions, and end-to-end semantics.

One might argue that a significant drawback to Active Names is that we modify the existing naming abstraction. Traditionally, Internet name resolution returns an IP address; the application uses the IP address to establish a socket connection to the end host. With Active Names, name resolution and retrieval of the specified resource is a single step that combines location and transport. A similar debate on the structure of naming has occurred in file systems and databases; the conclusion has been that it is dangerous to separate naming from use [43]. Superficially, it can seem simpler to have the naming system return an ID (IP address, inode number, or physical memory location) that is then used by the system to access the named resource. However, if the ID is visible to applications, the ID becomes hard state, something that must continue to be supported by the system even after the binding has changed or the reference has become invalid. Of course, to support legacy applications, the change to support unified naming and transport usually can be hidden inside an application-specific proxy; for example, web browsers can be configured to connect to a proxy that mediates the browser's interaction with the network.

The remainder of this paper covers these issues in more detail. We first discuss the strengths and weaknesses of Active Names by contrasting it to two popular alternatives. We next outline the architecture of our system; we then describe several applications we have built on top of our system. We conclude with related work and a summary of our contributions.

2 Background

To motivate our decision to provide extensibility via naming, we compare our approach to two popular alternatives, Active Networks and Active Services, that comprise extreme endpoints in this design space. At one extreme, by applying arbitrary computation on packets as they flow through routers, Active Networks can be completely general and transparent to end hosts; however, this transparency comes at a cost of both efficiency and in making it more difficult to express end-to-end semantics. At the other extreme, Active Services provides a framework for applications to execute arbitrary computation in the network; however, each application is free (even encouraged) to link with a different framework customized to its needs, making it more difficult to share extensions across applications. Active Names attempts to combine the best of both worlds; of course, our approach has its own set of limitations.

A principal advantage of our approach is that naming is at the top of the network protocol stack; by hijacking name resolution, we can gain control over (and therefore can extend) any network access. At the other end of the spectrum, hijacking packet processing inside routers likewise offers the ability to extend any network access. For example, consider an anycast service that routes client requests to the "best" of several different servers according to some selection criteria. In our system, such a new policy for server selection can be im-

plemented by mapping the service name to a program that selects the replica. Equivalently, the same policy could be implemented at the packet level inside programmable routers by mapping the name to a group address, and then dynamically routing packets to the desired server.

However, extending names offers simpler end-to-end failure semantics than is possible when extensibility is applied further down the protocol stack. Today, it is possible to build highly available services inside of a machine room [5, 45]. However, the end-to-end availability of wide-area services further depends on external factors such as power outages and whether packets can be routed from a client to the machine room. For example, routing pathologies can make a service appear unavailable to some clients even though the service is otherwise "up." To cope with these external factors, the service needs to be replicated at multiple geographic locations, each of which may fail independently. In the case of many read-only replicas, it is straightforward to redirect requests to a failed replica to any member of the group at multiple points in the protocol stack. However, for replicas that are writable, maintain state, or require authentication, the recovery protocol can require the coordinated activity of both the client and other replicas. For example, Bayou guarantees session consistency by restricting clients to bind only to replicas that have seen the client's updates [47]. Implementing session consistency correctly via transparent packet processing requires the network to maintain hard state about the behavior of the client; worse, this state largely duplicates what the client already has stored in its cache. Cheriton and Skeen [13] have cataloged numerous examples where failure recovery cannot be correctly implemented inside a transparent transport layer protocol. In our model, replica failures can be either reflected back to the client or handled transparently, under control of the program providing the binding between the name and the server.

Active Names is also more efficient than packet level filtering for those services that can be provided within our model. By interposing on connection setup, the overhead of programmability is typically paid once per connection, instead of once per packet. However, there are some services which cannot be provided above the packet layer and thus are not supported by our system; these include packet-level scheduling and resource allocation in routers and multi-host transparent packet filtering such as firewalls.

At the other extreme, some researchers have proposed customized application-level frameworks as a way of supporting application-specific computation inside the network. For example, the Active Services framework implements dynamically relocatable multimedia gateways; they suggest that different frameworks would be needed to support different applications [4]. Our approach differs from Active Services in that we are trying to provide a single, general-purpose framework capable of supporting the composition of a wide range of new services. Our goal is to allow Active Name modules to be developed and reused in a variety of contexts [31]; for instance, Active Services supports customizable filtering at the media gateways, but does not support customizable protocols for locating, managing, or communicating with the gateways, nor does the framework support dynamic installation of new implementations of client software. Except for the code to transform the multimedia stream to fit a limited link capacity, the client-gateway and gateway-server protocols in Active Services are fixed and non-extensible. By contrast, Active Names allows all aspects of service location and transport to be customized by the namespace owner; code is referenced by name, allowing us to use Active Names to locate and download new implementations of extension code whenever the namespace binding is changed.

3 Active Name Architecture

To support the functionality discussed above, four goals drive our architectural decisions. The architecture must (i) support customization and extensibility of each namespace, (ii) support composibility of different namespace customizations, (iii) support the efficient use of network resources, and (iv) support location-independent execution of namespace resolution.

Our core system is simple, and we will describe it by first providing an overview and then examining four key concepts of the design: the microkernel approach, location independent active name programs, namespace delegation, and the after-methods programming model. Because one of the major motivations of Active Names is to improve the performance of clients accessing services, our design is careful to allow room for several performance optimizations that complicate an otherwise straightforward design.

The core of the system is fully functional, and we have constructed a number of useful applications. The system is complete and stable enough for our own internal use. As detailed below, some aspects of security have not been fully integrated into the prototype.

3.1 Overview

A client that wishes to access a service constructs an Active Name for that service consisting of a *name* to resolve and the name of a *namespace program* to re-

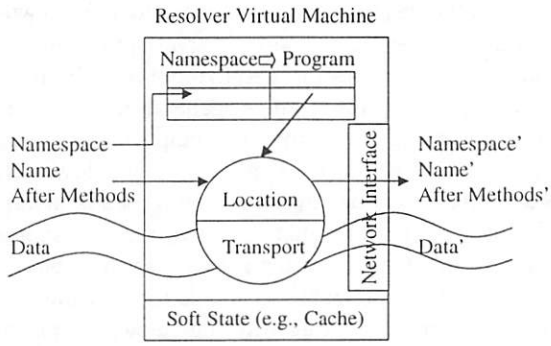


Figure 1: Active Names Architecture Summary.

solve it. The client then hands the name to the nearest *resolver*, which executes the namespace program to begin resolution of the name. Figure 1 illustrates the basic Active Names architecture.

A namespace program has two tasks: it must locate the next program to run and then transport data to that program. In doing so, namespace programs effectively establish a path through a series of namespace programs from request-source to reply-sink. Each program then acts as a filter that transports and transforms its input to its output.

A program locates the next program to run using two mechanisms. First, each Active Name is resolved recursively using a hierarchical delegation mechanism: a client specifies a root namespace program which partially resolves the name and determines which namespace program has jurisdiction over the remainder of the name; the root program then hands the partially-resolved name to the next namespace program, which continues the process. Second, to support composibility of services and to increase efficiency, the programming model follows a continuation passing style: as names are resolved the system constructs an *after-methods list* that uses a list of Active Names to describe a pipeline of services that will transport a result to its destination. Thus, once a name is recursively resolved to a service, rather than returning the service's output through the same call path used to resolve it, the leaf namespace program pops the top Active Name from the after-methods list and resolves that Active Name and the remaining after-methods list to transport the service's output to the client.

Resolvers provide the basic resource necessary to execute namespace programs, and they are distributed through the system. In principle, resolvers may be located anywhere, but in practice they are most useful when they are located at "interesting" points in the network: near clients, near servers, or near bottleneck network links. We envision a system that provides suitable resolver infrastructure at such points.

3.2 Extensibility

To support extensibility, the architecture follows the philosophy of providing a basic set of building blocks and allowing services and clients maximum freedom to customize the systems for their needs [1]. At minimum, each resolver must provide a loader for fetching and loading an active name program, safe execution of untrusted code, local soft state, and interfaces for communicating with and invoking programs on remote nodes.

For safe execution, our prototype relies on the Java-2 security system [53], but we could have just as easily chosen another mechanism such as hardware protection domains, or software fault isolation [52]. On top of this basic security mechanism, individual programs define policies for delegating namespaces they control and for accepting requests from other namespaces.

To provide a hook for Active Name programs to enforce security, the interface also provides as input a capability certificate that identifies the caller and which may grant a subset of the caller's rights to the callee. If the program is invoked from a remote node, the certificate will be authenticated via encryption techniques; if the program is called locally, the identity of the caller is guaranteed by the integrity of the local operating system. An Active Name program is free to use this information about the caller for access control. For example, a program could choose to run only on behalf of previously registered users. Similarly, if a program needs to enforce that its after-method is invoked, it grants the downstream program the right to reply to it but not the right to reply to the original caller. Certificates may also be required from the machines used to run the programs and after-methods, since a program's results should not be trusted unless it is run on trusted machines. We expect Active Names programs to leverage the work of other researchers in showing how to provide authentication and access control for mobile computation [3, 7]. We have implemented a prototype of such a certificate-based capability system, but we have not yet integrated this functionality into the Active Names prototype.

In a production system, nodes would enforce resource limitations using technology such as Jres [16]; such functionality is not implemented in our prototype.

3.3 Active Name Programs

To support efficient use of network resources, location-independent Active Name programs represent services and handle name resolution for them.

To run a namespace program, resolvers must first fetch and load the code for the program. Like other resources in our system, code is identified by the Active Name that describes how to locate it and transport it to the resolver. This allows us to use the Active Name system itself to load programs, which provides the ability to customize how to locate an available or nearby replica storing the program, to maintain version consistency, or to compress/decompress the binary. Of course, the recursion has to stop somewhere: a small number of initial programs (e.g., DNS, HTTP, etc.) are loaded onto each Active Name machine to bootstrap the system.

Given that Active Names programs can run in any resolver, an important question is determining where they should run. Each program is responsible for locating the data and computational resources it needs to complete its task, and each program works to minimize the cost of accessing these resources. For example, if a pipeline of programs needs data from a particular node, each step in the pipeline should take the request closer to the data. The details of how these decisions are made are namespace-dependent and can range from simple (e.g., the name being resolved includes the IP address of the node on which to run) to sophisticated (e.g., running a cost/benefit analysis comparing several different locations). Because location preferences are namespace dependent, one program does not typically know where the next prefers to run. Therefore a program generally invokes the next program locally; if the invoked program prefers to run somewhere else, it uses the remote execution interface to invoke the same program on a resolver node more to its liking.

3.4 Hierarchical Name Space

Active Names are organized hierarchically into *namespaces*, analogous to domains in the Domain Naming System (DNS) and directories in a UNIX file system. Names within a namespace can be, in turn, namespaces (subdomains in DNS or subdirectories in UNIX); they can also be terminal leaves in the naming tree (machines in DNS, files in UNIX). Each namespace has a program associated with it that is responsible for interpreting that portion of the namespace; this program is free to interpret the names within the namespace in any fashion it wants. A root namespace interprets all names. Each namespace has an owner with the right to determine the program bound to the namespace. The client, by default, is the owner of root. This allows the client to install a program to mediate how its names will be translated. For example, a PDA could install a root program to take whatever is returned by lower level name spaces and compress any images to fit the screen size [23].

To illustrate how delegation works, consider how our system implements the WWW namespace to support per-service naming and transport. Traditionally, users type web requests that specify a specific transport protocol (e.g., "http") along with the service name (e.g., "cnn.com"). But the transport used to communicate with a service should not be the concern of the end-user. In our framework, users simply name the service they wish to contact, and services specify the transport for names they control via the hierarchical namespace delegation mechanism. In particular, the root namespace sends web requests to the WWW-root Active Name program, which implements the WWW-root namespace. For bootstrapping (and by default), WWW-root delegates incoming requests to a series of Active Name programs that implement the default HTTP caching and transport protocol. But under the Active Names paradigm, rather than delegate resolution of all names to HTTP-default, the WWW-root namespace has the right to delegate portions of the WWW namespace to other Active Name programs according to any policy it chooses. The WWW-root's policy is to set these mappings as follows: the response for a request to a URL may include in its MIME header a directive specifying an Active Name program to be invoked for subsequent requests for which that URL is a prefix. For example, the reply to a request to `www.cs.utexas.edu/home/smith` can delegate the `www.cs.utexas.edu/home/smith/active/*` namespace, but it cannot delegate the `www.cs.utexas.edu/home/jones/*` namespace, the `www.cs.utexas.edu/*` namespace or the `www.cs.duke.edu/home/smith/*` namespace.

3.5 After-Methods

To support composibility and network efficiency in the transport of services, our programming model is to construct a chain of unidirectional filters from request source, through intermediate services, to reply sink. Intuitively, if an Active Name program acts as a layer in a protocol stack, each program should also provide a bidirectional pipe between the layer above it and the layer below, to provide a path for bytes to be sent between the client and the server. Each layer would then be able to filter the bytes sent on the connection as needed.

For efficiency reasons, we take a slightly more complicated approach. Frequently, an Active Name program is only a forwarding agent – it points to where the named resource can be found. In this case, it would be inefficient to treat the chain of Active Name programs as a pipe, forcing all bytes to traverse back through the chain of programs that led to the server; the inefficiency is particularly pronounced when the forwarding agent runs on a machine remote from both the client

and the server. Rather, our system uses a form of "multi-way RPC" based on a distributed continuation-style programming model: before passing control to the next namespace program to interpret the remainder of the name, the current namespace program bundles up its remaining work into an Active Name representing an "after-method" and prepends it to the list of after-methods created by earlier programs. The chain of after-methods is effectively a script of filters used to transport and transform the data being returned by the service once the name is fully resolved. For example, a program to compress data to increase network bandwidth would add the decompression routine as an after-method. Like other Active Name programs, these after-methods are free to run anywhere and subsequent programs may reorder the list.

3.6 API

At the API level, a namespace program takes a string (the remaining part of the name to be resolved), a reference to a data stream (the input to the service the name represents), and a list of after-methods (the Active Names of services needed to transport the result of this service to its destination.) The namespace program first determines which namespace program to call next by partially evaluating a name and then delegating further resolution to a sub-namespace or—if the namespace is a leaf and the name is fully resolved—by popping the top after-method from the after methods list.

Then, if the program wants additional work to be done with the result of the call, it adds the corresponding after-methods to the after methods list.

Finally, the program calls the next program with the partially resolved name, the remaining list of after-methods, and a data stream that comes from either i) passing the incoming data stream to the next program unchanged, ii) creating a new data stream by filtering the incoming data stream, or iii) creating a new data stream from local state (e.g., by reading data from a cache).

To be practical, our Active Names architecture must be able to be smoothly integrated with legacy clients, servers, and name databases. We accomplish this by using either a library or a proxy that provides default translations between legacy names and corresponding Active Names. For example, we provide a web proxy that allows unmodified browsers to use the Active Names system.

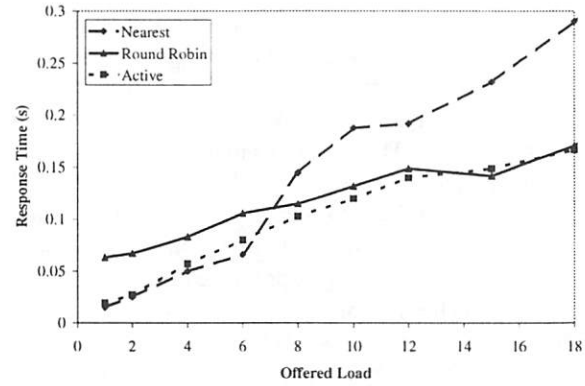


Figure 2: Replicated Service Access.

4 Applications

Given the Active Names framework described above, this section demonstrates the power of three key Active Name principles: extensibility, location independence, and composibility. First, we describe how Active Names support flexible end-to-end bindings between clients and replicated Internet services. Next, we show how Active Name filters are dynamically allocated to strategic wide-area locations to maximize client performance, and reduce consumed wide-area bandwidth and server load. Finally, we demonstrate the generality of the system by showing how individual Active Name extensions are composed together to provide significant performance benefits over any individual extension.

4.1 Extensibility

Active Names allow service-specific programs to account for any number of variables in choosing among server replicas, including client, server, and network characteristics. It is beyond the scope of this paper to determine the appropriate replica-selection policy for arbitrary services. However, we attempt to motivate the need for programmability in locating wide-area resources and the benefits of using end-to-end information for replica selection. We conducted the following experiment to demonstrate these points. For these measurements, between one and eighteen clients located at U.C. Berkeley attempted to access a service made up of two replicated servers, one at U.C. Berkeley and the second at the University of Washington. The clients use one of the three policies to choose among the replicas:

- *DNS Round Robin*: In this extension to DNS, a hostname is mapped to multiple IP addresses, and the particular binding returned to a client requesting hostname resolution is done in a round robin

fashion. Services employing DNS round robin achieve randomly load balanced access to replicas. In our experiments, we implement the Round Robin approach in Active Names by randomly choosing among available replicas.

- *Distributed Director*: This product from Cisco [15] executes specialized code in routers to allow services to register their current replica set. Requests (at the IP routing level) bound for a particular service, are automatically routed to the closest replica (as measured by hop count). While still not extensible, Distributed Director achieves geographic locality for service requests. In our experiments, we implement the Distributed Director approach in Active Names by always choosing the nearest replica.

- *Active Names*: With this instance of programmable replica selection, the program uses the number of hops (as reported by traceroute) from the client replica to determine the choice of replica. Replicas further away are less likely to be chosen than nearby replicas. However, this weighing is further biased by a decaying histogram of previous performance. Thus, if a replica has demonstrated better performance in the recent past, it is more likely to be chosen. For example, if replica A is 4 hops from a client, while a replica B is 5 hops away, A is randomly chosen 55% of the time. This probability is equally weighted with observed performance from the replicas. Thus, if performance histograms predict a 5-second access time for A and 4-second access time for B, based on performance alone, B will be chosen 55% of the time. Based on confidence in performance prediction and desire to localize traffic, different weights can be assigned to these two components. For our experiments, the components were weighed equally.

Figure 2 plots the average latency as a function of offered load as perceived by clients continuously requesting a 1 KB file from the replicated service. At low load, the proper replica selection policy is to choose the "nearest" replica at U.C. Berkeley. Thus, the Distributed Director policy shows the best performance at low load. However, as load increases, the U.C. Berkeley replica begins to become over-loaded, and the proper policy is to send approximately half the requests to the University of Washington replica. In this regime, high load at the U.C. Berkeley server offsets the cost of going across the wide area. Such load balancing is implemented by DNS round robin, which achieves the best performance at high load.

The simple Active Names policy is able to track the best performance of the two policies by accounting for distance and previous performance. At low load, both factors heavily bias Active Names toward the U.C. Berkeley replica. However, as load increases and performance at the U.C. Berkeley replica degrades, an increasing number of the requests are routed to the University of Washington, achieving better overall performance.

We do not purport our algorithm for replica selection to be optimal. However, it demonstrates the utility of programmable replica selection and the importance of using end-to-end performance measurements in choosing among wide-area replicas. While the above example simplisticly measures performance based on the latency for accessing fixed-size files, more sophisticated Active Names programs could account for the size of requested objects (e.g., optimizing latency for small objects and bandwidth for larger objects) or for the cost of dynamically generating content at the server (e.g., selecting strictly based on estimates of server load when making computationally-intensive requests). Only the end client has information about the *type* of request being generated, and thus only the client can use this information to influence replica selection in an end-to-end and application-specific manner. Schemes such as DNS round robin and Distributed Director are both too static and too far removed from the clients to utilize all relevant information in the replica selection process. For this type of application, Active Networks suffer from a similar lack of end-to-end information because of its focus on applying programs to individual packets in the middle of the network.

4.2 Location Independence

As discussed earlier, the proper way to present web content to a particular client depends upon its individual characteristics. For example, it makes little sense to transmit a 200 KB 1024x768 color image to a handheld device with a 320x200 black-and-white screen behind a wireless link. To address this mismatch, one current approach [23] is to mediate client access through web proxies. These proxies retrieve requested resources and dynamically distill the content to match individual client characteristics, e.g., by shrinking a color image and converting it to black-and-white.

At a high level, clients name a web resource but would like the resource transformed based on client-specific characteristics. This model fits in well with Active Names. Clients specify the name of a resource (e.g., a URL retrieved by an HTTP namespace program) and an after-method that specifies the distillation program to be applied on the resource once it is located. The

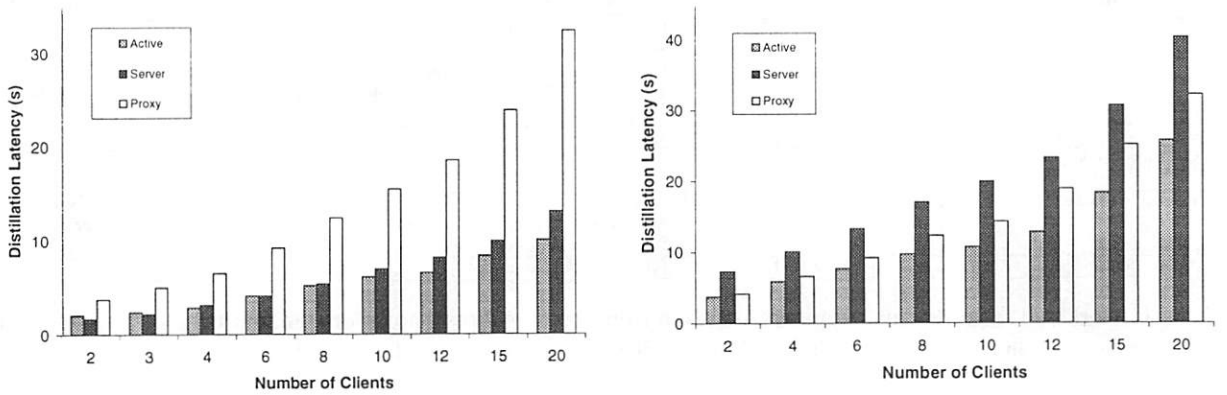
distillation program ensures that the object returned to the client will match its characteristics. A benefit of using Active Names to encapsulate distillation is the ability to flexibly place the transformation of a requested resource at arbitrary points in the network. For example, if the network path between a server and a proxy is congested, it may not make sense to transmit a large image over the congested network to perform a transformation that greatly reduces the size of the image. In a classic function versus data-shipping tradeoff, it is usually more efficient to perform the transformation at the server and then to transmit the smaller image to the proxy (or directly to the client). Conversely, if the transformation function is expensive, a fast network connection is available, and the server CPU is heavily loaded, then it is often more efficient to transmit the larger image to a proxy (or client) where more CPU cycles are available. Thus, the location-independent programs that comprise Active Names allow for flexible evaluation of function versus data shipping, trading off network bandwidth for computation time.

To demonstrate the above points, we implemented distillation within the Active Names framework and ran the following experiment to evaluate its utility. A client at U.C. Berkeley requests, through a local proxy, an image located at Duke University. This request is made under a number of different circumstances. The first variable is the place where distillation takes place. Active Name resolvers are available at both U.C. Berkeley and Duke so distillation can take place at either location. Three different policies are evaluated in choosing the distillation point: i) Statically assigning distillation at the proxy, the current approach to distillation, ii) Statically assigning distillation to the server, transmitting a smaller image across bottleneck wide-area links, and iii) an active approach where distillation is randomly assigned biased by estimates of end-to-end distillation cost at both the proxy and server sites.

In the active scheme, a Active Name after-method caches CPU load information at both the server and the proxy. Cache values are considered fresh for one minute. When cached load information expires, a separate thread is spawned to refresh cache information (the program responsible for maintaining load information, being an Active Name, can run either locally or remotely). The distillation Active Name program uses this load information, in addition to an estimate of the cost of unloaded distillation based on the size of the image, to calculate distillation cost at both the server and the proxy. The program also calculates the cost of transmitting either the full or distilled image to the proxy to arrive at an end-to-end cost of distilling the image at the two locations. The distiller uses this information to bias a random selection of the location for distillation. Thus, if it is estimated that it will be twice as expensive to perform distillation at the proxy, the chance of performing proxy distillation will be one in three.

Another variable considered in our experiments is the load on the server machine at Duke. In one case, the Name Resolver at Duke University runs on an otherwise unloaded machine. In another, the resolver must compete with ten CPU-intensive processes. The load on the server CPU will impact the placement of the distillation program. A third variable in our experiment is the dynamically changing available bandwidth between U.C. Berkeley and Duke (located at opposite ends of a continent). For this experiment, only the first two variables are modified. Available bandwidth is kept constant (as much as possible) by running the experiment late at night. In the future, we plan to investigate the use of SPAND [46] to estimate available bandwidth between two wide-area sites, and to use this information to more intelligently choose the location of distillation. The Active Name resolvers (including all distillation code) are compiled and run with the Java Development Kit, version 1.2 beta 4. The target image is 59 KB and is distilled to 9 KB. Distillation of the image consumes approximately 1 second of CPU time.

Figure 3 graphs the client-perceived latency of retrieving distilled versions of the target Jpeg image as a function of the number of clients simultaneously requesting the image from Duke for the three evaluated policies (static proxy, static server, active). Figure 3(a) shows performance in the case where the server is unloaded, while Figure 3(b) addresses a heavily loaded server, competing with ten CPU-intensive processes. Figure 3(a) shows that, at low levels of offered load (few simultaneous clients), unilaterally placing distillation at the server produces the best results because a smaller amount of data (9K versus 59K) is shipped across the wide area. However, as the number of clients increases, the server at Duke becomes overloaded and the performance degrades relative to the active policy that intelligently allocates distillation of a random percentage of the requests to the proxy. Figure 3(b) shows that, at low levels of offered load and high server CPU load, it is beneficial to place distillation at the proxy site. In this case, distillation at the server is an expensive enough operation to justify the larger long-haul transmission costs. However, as offered load increases, the active policy of splitting requests between the server and the proxy sites once again outperforms the static policy because the single processor at the proxy becomes overloaded.



(a) Unloaded Server

(b) Loaded Server

Figure 3: Mobile Distillation Performance.

4.3 Composibility

A key design goal of Active Names is composibility. Not only should applications be able to inject extensions into the network, but they should be able to combine these extensions to provide new services and optimize existing ones. In this subsection, we first examine the costs composibility imposes on the system. We then study the benefits composable extensions bring to a key problem: web caching. Caching is a key technique for reducing both long-haul bandwidth and client-perceived latency. Table 1 breaks down the reason for web caching's relatively low hit rates that hang stubbornly near or below 50% [19, 27]. This table suggests that composing different extensions may be a key technique for addressing the web caching problem. Moreover, these extensions are likely to be provided and implemented by a number of different entities, ranging from clients, to service providers, to third party software vendors. The experiments in this section demonstrate how the Active Name framework is utilized to compose multiple independent extensions, resulting in greater end-to-end performance gain than available from any single approach.

Our continuation-passing architecture imposes the overhead of indirection through the "after-methods" script when one Active Name program transfers control to another. On a Sun Ultra-10 with a 300MHz UltraSparc-II process running JDK 1.2 fcs, it takes 3.2 μs for one Active Name program to call another and return via this mechanism, compared to 0.2 μs if it were allowed to make the procedure call directly. Although this is more than an order of magnitude worse, the performance is sufficient to support the composition of relatively coarse-grained services such as we envision.

To determine whether composibility is worth this cost, we compare the cost the composition of several server-initiated customizations against the cost of a client-initiated customization. The semantics of the sample service we implement are that when a client fetches a base page, the service i) uses the server-side include interface to update the page for the current request, ii) randomly selects two out of eight candidate "banner ad" inline images, repeating the random selection on each viewing of the page, and iii) logs the provided cookies the advertisements selected for each request. We implement these server semantics in two ways. First, we construct them using standard mechanisms that run at the server: the server uses server-side includes to update the page and to execute a perl program that randomly selects advertisements to include on the page; we do not add additional logging to that already provided by the HTTP server. Second, we implement a version of the service by delegating a portion of the HTTP namespace to a set of three Active Name programs (provided by the service). The default HTTP namespace delegates control of all HTTP requests destined to `www.cs.utexas.edu/users/anonymized/service` to a "controller" Active Names program that alters the return path for such requests through a "ssi" Active Name program that provides server side include semantics and through an "advertise" Active Name program that does advertisement rotation and logging. In the first configuration, all requests for the base page must go to the server; requests for the inline images may be cached. In the second configuration, once the delegation Active Name programs have been installed, both the base page and the inline images may use the cache because cached results will pass through the ssi and the advertise Active Names programs before being returned to the client.

For the client-initiated customizations, clients use

Source of Miss	Fraction of Requests	Available Approaches	Client/Server Initiated
Compulsory	19%-30%[48] 45%(*ISP*)	Prefetching [28, 35, 44]	either
		Server replication [50] or push caching [30]	server
		Increase number of clients sharing cache system [11, 19, 27, 48]	client
		Transcoding [23, 4], compression and delta-encoding[40]	client or either
Consistency verify (unmodified)	10%(*ISP*) 2-7%[19] 4-13%[6]	Server-driven consistency [37, 56]	server
Consistency miss	0-4%[19]	delta-encoding[40]	either
Dynamic (cgi or query)	21%(*ISP*) 0-34%[38]	Active cache [12], HPP [18]	server
		TREC [49]	server
Pragma: no-cache	9%(*ISP*) 5.7-7.2%[27]	Hit logging	server
		Active cache [12], function-shipping	server
		Server-driven consistency [56]	server
Redirection	3.7%(*ISP*)	Server selection/anycast [9, 57]	server

Table 1: Workload requirements. Numbers are taken from the literature as noted or from our trace of a large ISP that serves seven million requests containing 65.4GB to 23080 clients over a six-day period .

Active Names to customize their namespace to transcode images sent across a slow modem link. Because clients control their own namespace, adding this transformation to the pipeline is straightforward. The main subtlety is that clients cannot store the distilled images in the standard HTTP cache lest one client's mapping of the URL to the customized image disturb other clients. Rather than cache such results in the HTTP namespace, the client caches such results in the "distiller" namespace instead.

Our experimental set up consists of three machines. The client, a 133 MHz Pentium machine running Microsoft NT3.5 and Sun JDK 1.2 beta 3, communicates with the proxy, a 300MHz Sun UltraSPARC-II machine running Solaris 5.6 and JDK 1.2 fcs, over a 28.8 kbit/s modem. Both the client and proxy run the Active Name framework. The service being tested is hosted on a departmental web server running on a dual-processor Sun SPARCServer 1000e running Solaris 5.5.1 and the Netscape Enterprise Server 3.0(J) HTTP server. The proxy and server are connected by a department-wide switched 100 Mbit/s Ethernet.

The base page and its header are between 657 bytes and 2393 bytes (depending on where customization occurs) and the advertisement banner images range in size from 8421 to 16928 bytes before distillation and from 2595 to 4365 bytes after distillation. The JAR files containing the server's controller, advertise, and ssi customization programs are 2622, 4700, and 3274 bytes, respectively. We begin the experiment with cold caches, except that we fetch two unrelated HTTP documents through the system to cause the JVMs to pre-load most of the basic classes associated with the system's standard HTTP data path, and we fetch two unrelated image files to cause the proxy to load the client's distiller Active Name program.

Figure 4 shows four cases representing the permutations of distillation (on/off) and server customization (on/off). Our client driver program uses the Active Name system running at the client to fetch the base document and then, using parallel connections, to fetch all inline images specified by the base document. After the driver receives each page and associated inline images, it pauses five seconds and repeats the process. The variation in response times from request to request is caused by cache hits and misses to the base page and the randomly selected inline images.

With respect to server customizations, there are three phases to consider. On the first request, no delegation has yet been specified to the client's Active Name system, so the "Server: on" performance closely matches the "Server: off" performance. Reacting to the delegation directive in the first request, the client's Active Name system spawns a background thread to download and install the specified customizations. This background thread is active during the second phase of the experiment—request two for the case when distillation is turned off and requests two and three when distillation is on. As a result, performance for these requests is noticeably worse under server customization than for the standard case. In the third phase—after request three—the client has installed the server customization into its namespace and thus no longer needs to go to the server for advertisement rotation, hit logging, or server side include expansion. Performance is now significantly better under the customized version (modulo cache hits to the inline images). For example, as Figure 4(a) indicates, after the cache is warm and when the inline images are hits, the "Server: on" case provides response times under 0.26 s while the other case requires over 1.3 s per request. In this situation, the Active Names system provides a 5-fold performance improvement. This result is particularly significant in light of human factors studies that suggest that driving computer response time from about a second to significantly less than a second may result in more than a linear increase in user productivity as the system becomes truly interactive [32, 10].

Figure 4 also shows that distillation significantly improves performance for the initial series of requests, and

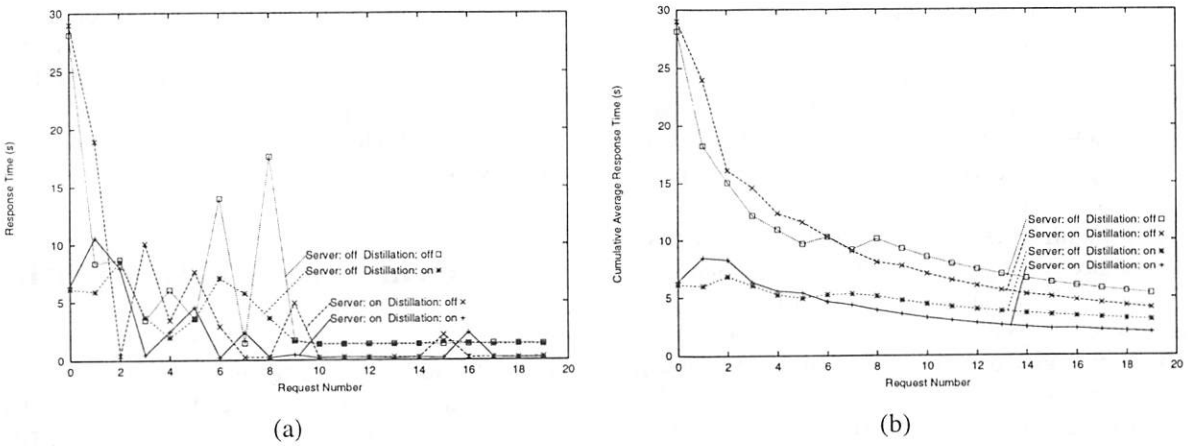


Figure 4: (a) Per-request and (b) cumulative average performance of customizations of the HTTP namespace.

makes little difference once the images are cached at the client. For example, when server customization is turned on, the five most expensive requests require an average of 15.3 s without distillation, whereas the five most expensive requests averaged 6.67 s under distillation, a speedup of 2.3. Without server customization, the five most expensive requests average 14.1 s and 6.39 s when distillation is off or on respectively, for a speedup of 2.2 for distillation.

Other researchers have noted the advantage of server-controlled caching [12] and distillation [23]. The above experiment suggests that a combination of server and client customizations may be particularly effective. On average for the 20-request sequence, the combination outperforms the distillation-only case by 50% and it outperforms the server-only case by 104%.

5 Related Work

As discussed in Section 2, Active Names are inspired by and provide an alternative to related research in Active Networks and Active Services. Also closely related to Active Names are Transaction Processing monitors [25] (TP monitors). TP monitors provide functionality similar to Active Names for access to databases. The TP monitor directs transactions to *resource managers*, accounting for load on machines, the RPC program number, and any affinity between client and server. Resource managers are usually SQL databases but can be any server that supports transactions. While Active Names and TP monitors target a number of similar applications, Active Names provides a more general environment for programmable access to wide-area resources. In contrast, TP monitors tend to be more static and more closely associated with the service.

Active Names are also related to the Intentional Name System [2]. Similar to the Active Name approach, Intentional Names take the stance that applications use naming to describe what they are looking for, as opposed to where to find it. Intentional Names utilize declarative style data structures for maintaining attribute-value pairs used to bind a user-specified name to an appropriate instance of the target resource. However, Intentional Names are not programmable, and thus difficult to specialize to individual application requirements, and are also not designed to operate in wide-area environments, targeting single administrative domains. Further, while Intentional Names do support flexible resource location, they do not incorporate efficient transport of resources back to clients.

Other systems have also supported programmable name translation. For example, object-oriented systems such as Smalltalk have long provided application control over the binding between caller and callee. The Mercury RPC system did the same in the mid-1980's for a distributed client-server environment.

Current wide-area computing research proposals, such as Globe [51], Globus [20], and Legion [29], propose a number of schemes for locating computational resources across the wide area. These proposals are orthogonal to our work as any of them could be incorporated within the extensible Active Names framework. Ninja [26] proposes a pipelined data flow model for composing services in a clustered and/or mobile environment. While this model is attractive, it does not to date address dynamic migration of computation, as we have demonstrated is crucial for performance. A recent proposal [17] for implementing URN's advocates leveraging DNS and rewriting of names through regular-expression matching in an iterative manner to locate wide-area resources. This scheme could also be imple-

mented more generally within the Active Names framework with namespace programs responsible for name rewriting and namespace delegation.

Prospero [41] also supports extensible naming to support mobile hosts and the integration of multiple wide-area information services (e.g., WAIS and gopher). Prospero allows users to customize their own namespaces, grouping related information (from an individual's perspective) together. However, customization code runs on the client. Relative to Prospero, our work demonstrates the utility of location-independent and portable programs for name resolution. Programmability in Active Names is similar to Smart Clients [57]. Smart Clients retrieve service-specific code into the client to mediate access to a set of server replicas. Active Names are more general than Smart Clients, with location independent code able to run anywhere in the system, allowing for the deployment of a broader range of applications.

Application-layer Anycasts [9], Nomenclator [42], and Query Routing [36] also allow for resource discovery and replica selection. Anycasts allow a name to be bound to multiple servers, with any single request transmitted to a single replica according to policy in routers or end hosts. Nomenclator uses replicated catalogs with distributed indices to locate wide-area resources. The system also integrates data from multiple repositories for heterogeneous query processing. Query Routing uses compressed indexes of multiple resources and sites to route requests to the proper destination. These approaches show promising results and should, once again, fit well within our extensible framework.

Active Caches [12] allow for customization of cache content through Java programs similar to our extensible cache management system. With Active Caches, however, retrieved data files contain programs, with the cache promising to execute the program (which may change the contents of the file) before returning the data to the client. On the other hand, our extensible cache management system uses service-specific programs to mediate all accesses to a service. This approach is more general, allowing, for example, the program to manage local cache replacement policy or to perform load balancing on a cache miss. We use a technique similar to Active Caches for delegating programs to individual names, but in keeping with the namespace paradigm, we allow parent directories to control the delegation of entire subdirectories rather than doing delegation on an object-by-object basis. Note that this approach of associating a program with each level of a hierarchical namespace is not new. The HP Brevix and MIT Exokernel research file systems, for example, have examined allowing users to define application-specific programs for each directory in a file system [22, 33]. A directory's program is completely responsible for managing the bits stored inside the directory; for example, this allows applications to customize on-disk data structures to optimize for application-specific reference patterns (e.g., storing HTML files with cross-links in the same disk cylinder).

6 Conclusion and Future Work

This paper has described a framework supporting extensibility for wide-area distributed services through the introduction of location-independent programs that interpose on the naming interface. These Active Names can, for example, customize how a service is located and how its results are transformed and transported back to the client. Our approach is compared to existing schemes for introducing programmability into the network such as Active Networks and Active Services. The paper then describes the implementation of the Active Name prototype and illustrates its utility through a number of sample services, including replicated service selection and mobile distillation of service content. In each case, end-to-end application performance information is leveraged to match or exceed existing static approaches. The need for composibility is illustrated through Internet service access that incorporates extensions from multiple sources. Our results show that Active Name extensions can offer up to a five-fold performance improvement relative to existing static approaches, and that it is necessary to compose multiple extensions to achieve this benefit: no single extension achieves comparable performance.

References

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A New Kernel Foundation For UNIX Development. In *Proceedings of the 1986 USENIX Summer Conference*, pages 93–112, June 1986.

[2] William Adjie-Winoto, Ellio Schwartz, and Hari Balakrishnan. An Architecture for Intentional Name Reseolution and Application-level Routing. Work in Progress, February 1999.

[3] D. Scott Alexander, William A. Arbaugh, Angelos D. Keromytis, and Jonathan M. Smith. Safety and Security of Programmable Network Infrastructures. *IEEE Communications Magazine*, 36(10):84–92, 1998.

[4] Elan Amir, Steven McCanne, and Randy Katz. An Active Service Framework and its Application to Real-Time Multimedia Transcoding. In *Proceedings of SIGCOMM*, September 1998.

[5] Thomas E. Anderson, David E. Culler, David A. Patterson, and the NOW Team. A Case for NOW (Networks of Workstations). *IEEE Micro*, February 1995.

[6] Martin F. Arlitt and Carey L. Williamson. Web Server Workload Characterization: The Search for Invariants. In *Proceedings of the ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 126–137, May 1996.

[7] Eshwar Belani, Amin Vahdat, Thomas Anderson, and Michael Dahlin. The CRISIS Wide Area Security Architecture. In *Proceedings of the USENIX Security Symposium*, San Antonio, Texas, January 1998.

[8] Time Berners-Lee, Robert Cailliau, Jean-Francois Groff, and Bernd Pollermann. World Wide Web: The Information Universe. In *Electronic Network: Research, Applications, and Policy*, number 1 in 2, Spring 1992.

[9] S. Bhattarcharjee, M. Ammar, E. Zegura, V. Sha, and Z. Fei. Application-Layer Anycasting. In *Proceedings of IEEE Infocom*, April 1997.

[10] J.T. Brady. A Theory of Productivity in the Creative Process. In *IEEE CG&A*, May 1986.

[11] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. On the Implications of Zipf's Law for Web Caching. Technical Report 1371, University of Wisconsin, April 1998.

[12] Pei Cao, Jin Zhang, and Kevin Beach. Active Cache: Caching Dynamic Contents on the Web. In *Proceedings of Middleware*, 1998.

[13] David Cheriton and Dale Skeen. Understanding the Limits of Causally and Totally Ordered Communication. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 44–57, December 1995.

[14] Stuart Cheshire and Mary Baker. Internet Mobility 4x4. In *Proceedings of the ACM SIGCOMM'96 Conference*, August 1996.

[15] Cisco. Distributed Director. `http://www.cisco.com/warp/public/751/distdir/technical.shtml`, 1997.

[16] Grzegorz Czajkowski and Thorsten von Eicken. JRes: A Resource Accounting Interface for Java. In *Proceedings of 1998 ACM OOPSLA Conference*, October 1998.

[17] Ron Daniel and Michael Mealling. Resolution of Uniform Resource Identifiers using the Domain Name System. Internet Draft, see `http://www.acl.lanl.gov/URN/naptr.txt`, May 1997.

[18] Fred Douglis, Antonio Haro, and Michael Rabinovich. HPP: HTML Macro-Preprocessing to Support Dynamic Document Caching. In *Proceedings of the 1997 Usenix Symposium on Internet Technologies and Systems*, Monterey, California, December 1997.

[19] Brad Duska, David Marwood, and Michael J. Feeley. The Measured Access Characteristics of World Wide Web Client Proxy Caches. In *Proceedings of the 1997 Usenix Symposium on Internet Technologies and Systems*, Monterey, California, December 1997.

[20] S. Fitzgerald, I. Foster, C. Kesselman, G. von Laszewski, W. Smith, and S. Tuecke. A Directory Service for Configuring High-Performance Distributed Computations. In *Proc. 6th IEEE Symp. on High-Performance Distributed Computing*, pages 365–376, 1997.

[21] Marc E. Fiuczynski, Vincent K. Lam, and Brian N. Bershad. The Design and Implementation of an IPv6/IPv4 Network Address and Protocol Translator. In *Proceedings of the 1998 USENIX Conference*, June 1998.

[22] Martin Fouts, Tim Connors, Steve Hoyle, Bart Sears, Tim Sullivan, and John Wilkes. Brevix design 1.01. Technical Report HPL-OSR-93-22, HP Laboratories, April 1993.

[23] Armando Fox, Steven Gribble, Yatin Chawathe, and Eric Brewer. Cluster-Based Scalable Network Services. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, Saint-Malo, France, October 1997.

[24] James Gosling and Henry McGilton. The Java(tm) Language Environment: A White Paper. `http://java.dimensionx.com/whitePaper/java-whitepaper-1.html`, 1995.

[25] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[26] Steve Gribble, Matt Welsh, Eric Brewer, and David Culler. The MultiSpace: an Evolutionary Platform for Infrastructural Services. In *Proceedings of the 1999 Usenix Technical Conference*, June 1999.

[27] Steven D. Gribble and Eric A. Brewer. System Design Issues for Internet Middleware Services: Deductions from a Large Client Trace. In *Proceedings of the 1997 Usenix Symposium on Internet Technologies and Systems*, Monterey, California, December 1997.

[28] J. Griffioen and R. Appleton. Automatic Prefetching in a WAN. In *IEEE Workshop on Advances in Parallel and Distributed Systems*, October 1993.

[29] A. Grimshaw, A. Nguyen-Tuong, and W. Wulf. Campus-Wide Computing: Results Using Legion at the University of Virginia. Technical Report CS-95-19, University of Virginia, March 1995.

[30] James Gwertzman and Margo Seltzer. World-Wide Web Cache Consistency. In *Proceedings of the 1996 USENIX Technical Conference*, pages 141–151, January 1996.

[31] Norm C. Hutchinson and Larry L. Peterson. The x-Kernel: An Architecture for Implementing Network Protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, January 1991.

[32] IBM. *The Economic Value of Rapid Response Time*, pages 11–82. Number GE20-0752-0. White Plains, N.Y., 1982.

[33] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Hctor M. Briceo, Russell Hunt, David Mazires, Thomas Pinckney, Robert Grimm, John Jannotti, and

Kenneth Mackenzie. Application Performance and Flexibility on Exokernel Systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, October 1997.

[34] Eric Dean Katz, Michelle Butler, and Robert McGrath. A Scalable HTTP Server: The NCSA Prototype. In *First International Conference on the World-Wide Web*, April 1994.

[35] T. Kroeger, D. Long, and J. Mogul. Exploring the Bounds of Web Latency Reduction from Caching and Prefetching. In *Proceedings of the 1997 Usenix Symposium on Internet Technologies and Systems*, December 1997.

[36] P. Leach and C. Weider. Query Routing: Applying Systems Thinking to Internet Search. In *Proceedings of the Sixth Workshop on Hot Topics in Operating Systems*, pages 82–86, Cape Code, MA, 1997.

[37] C. Liu and P. Cao. Maintaining Strong Cache Consistency in the World-Wide Web. In *Proceedings of the Eighteenth International Conference on Distributed Computing Systems*, May 1997.

[38] S. Manley and M. Seltzer. Web Facts and Fantasy. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, 1997.

[39] P. Mockapetris and K. Dunlap. Development of the Domain Name System. In *Proceedings SIGCOMM 88*, April 1988.

[40] Jeffrey Mogul, Fred Douglis, Anja Feldmann, and Balachander Krishnamurthy. Potential Benefits of Delta Encoding and Data Compression for HTTP. In *Proceedings of ACM SIGCOMM*, pages 181–194, August 1997.

[41] B. Clifford Neuman. Prospero: A Tool for Organizing Internet Resources. In *Electronic Networking: Research, Applications and Policy*, pages 30–37, Spring 1992.

[42] Joann Ordille and Barton P. Miller. Distributed Active Catalogs and Meta-Data Caching in Descriptive Name Services. In *IEEE International Conference on Distributed Computing Systems*, pages 120–129, May 1993.

[43] John Ousterhout. *CMU Computer Science: A 25th Anniversary Commemorative*, chapter The Role of Distributed State. ACM Press, 1991.

[44] V. Padmanabhan and J. Mogul. Using Predictive Prefetching to Improve World Wide Web Latency. In *Proceedings of the ACM SIGCOMM '96 Conference on Communications Architectures and Protocols*, pages 22–36, July 1996.

[45] David A. Patterson, Garth Gibson, and Randy H. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proceedings of the 1988 ACM Conference on Management of Data (SIGMOD)*, pages 109–116, Chicago, IL, June 1988.

[46] Srinivasan Seshan, Mark Stemm, and Randy H. Katz. SPAND: Shared Passive Network Performance Discovery. In *Proc. 1st Usenix Symposium on Internet Technologies and Systems (USITS '97)*, December 1997.

[47] Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, Mike J. Spreitzer, and Carl H. Hauser. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 172–183, December 1995.

[48] R. Tewari, M. Dahlin, H. Vin, and J. Kay. Design Considerations for Distributed Caching on the Internet. In *Proceedings of the Twentieth International Conference on Distributed Computing Systems*, May 1999.

[49] Amin Vahdat and Thomas Anderson. Transparent Result Caching. In *Proceedings of the 1998 USENIX Technical Conference*, New Orleans, Louisiana, June 1998.

[50] Amin Vahdat, Thomas Anderson, Michael Dahlin, Eshwar Belani, David Culler, Paul Eastham, and Chad Yoshikawa. WebOS: Operating System Services for Wide-Area Applications. In *Proceedings of the Seventh IEEE Symposium on High Performance Distributed Systems*, Chicago, Illinois, July 1998.

[51] M. van Steen, F.J. Hauck, P. Homburg, and A.S. Tanenbaum. Locating Objects in Wide-Area Systems. In *IEEE Communications Magazine*, pages 104–109, January 1998.

[52] Robert Wahbe, Steven Lucco, Thomas Anderson, and Susan Graham. Efficient Software-Based Fault Isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 203–216, December 1993.

[53] Dan S. Wallach, Dirk Balfanz, Drew Dean, and Edward W. Felten. Extensible Security Architectures for Java. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 116–128, Saint-Malo, France, October 1997.

[54] David Wetherall, Ulana Legedza, and John Guttag. Introducing New Network Services: Why and How. In *IEEE Network Magazine, Special Issue on Active and Programmable Networks*, July 1998.

[55] Yahoo. My Yahoo. `http://my.yahoo.com`, 1996.

[56] J. Yin, L. Alvisi, M. Dahlin, and C. Lin. Using Leases to Support Server-Driven Consistency in Large-Scale Systems. In *Proceedings of the Nineteenth International Conference on Distributed Computing Systems*, May 1998.

[57] Chad Yoshikawa, Brent Chun, Paul Eastham, Amin Vahdat, Thomas Anderson, and David Culler. Using Smart Clients to Build Scalable Services. In *Proceedings of the USENIX Technical Conference*, January 1997.

Person-level Routing in the Mobile People Architecture

Mema Roussopoulos Petros Maniatis Edward Swierk
Kevin Lai Guido Appenzeller Mary Baker
Department of Computer Science
Stanford University
Stanford, California, 94305
{mema, maniatis, eswierk, laik, appenz, mgbaker}@cs.stanford.edu, http://mosquitonet.stanford.edu/

Abstract

Ubiquitous network connectivity for devices does not automatically imply continuous reachability for people. People move from place to place and switch from one network device to another. As a result, phones ring in empty offices, email cannot reach most cell phones, and spam clogs expensive, low-bandwidth links to laptops. Whereas existing mechanisms have addressed host mobility or the mobility of people within one network, few have allowed *people*, the ultimate and most important endpoints of communication, to roam freely, without being constrained to one location, one application, one device, or one network.

We have designed the Mobile People Architecture (MPA) to maintain *person-to-person reachability*. The central component of MPA is a *person-level router* called the *Personal Proxy*. It tracks a mobile person's location, accepts communications on his behalf, converts them into different application formats according to his preferences, and forwards the resulting communications to him. In contrast to similar systems, the Personal Proxy protects the user's privacy, is easily extensible to new network devices and applications, and has been deployed with no modifications to the existing network and telecommunications infrastructure. In this paper, we describe the design, implementation, and preliminary evaluation of our prototype Personal Proxy, a service that integrates Internet and telephone communication and addresses the need for person-to-person reachability.

1 Introduction

One of the defining trends of this decade has been the explosive growth of the Internet and other communication networks. People have access to a growing number of networks (e.g., Internet, cellular, pager) on a growing number of devices (e.g., personal digital assistants, cell phones, smart cards) at a growing number of locations (e.g., work, home, on the road). Unfortunately a growing problem for these people is maintaining reachability: as network devices, applications, and accessible locations proliferate, it becomes less likely that other people (*correspondents*) can get in touch with a mobile person at any particular time. For example, a correspondent might not have the mobile person's hotel phone number, or the correspondent's email application might not interoperate with the mobile person's phone. We therefore believe that there is a need for a *person-level router* that meets the following goals:

Maintain person-to-person reachability. The router should direct the correspondent's communications to the mobile person, regardless of whether the two participants have direct access to the same kind of network, device, or application.

Protect the mobile person's privacy. To route communications to a mobile person, a person-level router must track the devices and applications through which the person is currently reachable. The router should not reveal this tracking information, whether current or historical, because it could be used to deduce the person's location and compromise his privacy. In addition, receiving unwanted messages is also an invasion of privacy. Many applications, such as those in many phone systems, have no way to deliver high priority communication intrusively while delivering low priority communication unintrusively. Users should be able to have all their incoming communications prioritized and filtered according to their preferences.

Extend easily to new network devices and applications. Given the rate at which communication

networks, devices and applications proliferate, it is essential that a person-level router be easily extensible.

Be deployable without modifying existing infrastructure. Considering the fast pace at which new networking technologies develop, a communication system that requires changes to the existing network and telecommunications infrastructure is difficult to deploy and runs the risk of becoming obsolete before it is widely available. A person-level router must be easily and rapidly deployable to benefit the greatest number of people.

In this paper, we describe the design and implementation of the *Personal Proxy*, a person-level router which we believe meets all of these goals. In Section 2, we give an overview of the Mobile People Architecture (MPA), which includes the Personal Proxy. In Section 3, we describe the Personal Proxy in detail. In Section 4, we evaluate our prototype using our design goals. In Section 5, we describe related work. Finally, in Section 6 we conclude.

2 Architecture Overview

In this section, we describe how person-level routing fits into the overall picture of networking and argue that the Mobile People Architecture [MRS+99] is a logical extension of the current model of networking.

Networking systems are traditionally organized using a layering model composed of Application, Transport/Network, and Link layers (Figure 1). This model is useful in clearly defining the responsibilities and restrictions of software that exists at each level. To be implemented fully, a layer needs a naming scheme, a way to resolve those names, and a way to route communications.

The *Name Types* column of Figure 1 shows the naming scheme that Internet email uses at each layer. Some examples of names are shown in the *Packet Headers* column. These naming schemes usually mandate that the names are unique and change infrequently. In addition, each layer in the figure has a protocol to map its names to lower-layer names (the *Name Lookup* column in Figure 1). This mapping facilitates routing a communication to its destination.

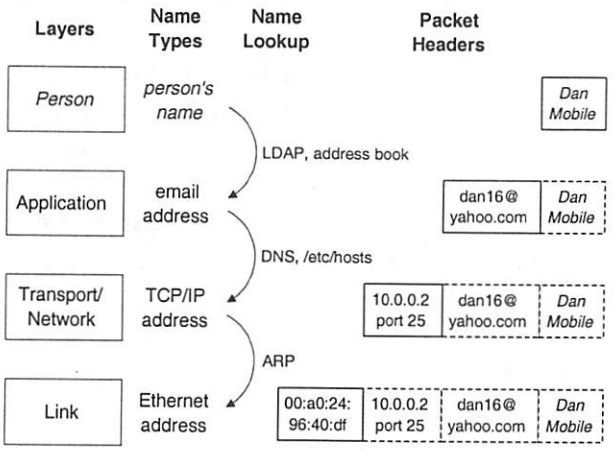


Figure 1: The Layering Model. We show the traditional networking layers, extended with the *Person* layer. *Name Types* shows examples of the kinds of names used at each layer. *Name Lookup* shows some methods of mapping names from each layer to names in the next lower layer. *Packet Headers* shows examples of actual names at each layer, and their relative locations in a typical email packet.

To model the full process of person-to-person communication, we need to extend the model to include people (the new *Person* layer, as shown at the top of Figure 1), since most important communication is ultimately from one person to another, rather than merely from one device to another. Currently, the Person layer is implemented in an ad hoc, non-unified way. People are not always named in a unique way, although a name or nickname is often unique among those with whom a person communicates frequently. These names (e.g., Dan Mobile) are resolved into application-specific addresses (ASAs, e.g., dan16@yahoo.com) using a directory service (e.g., LDAP [WKH97]), an address book, or simply from a person's memory.

Consequently, applications have difficulty delivering communication to people who move from one application-specific address to another. For example, if Dan Mobile stops using his cell phone because he entered his office building—where land-line phones are cheaper and have better quality—and starts using his office telephone, he might not receive Jane Sender's phone call in a timely manner. Furthermore, if Dan is temporarily unavailable by phone but is reachable by email, Jane, who may not be near an email terminal, cannot communicate with him until he is available by phone. Unfortunately, even in the case where Dan's cellular carrier

allows him to forward his missed phone calls to his office email address, Dan will still be unreachable when he moves from his office to a conference room. The problem is that Jane cannot identify Dan in a way that is independent of how he is reachable.

The solution is to create a unified implementation for the Person layer. Such an implementation needs to name people, map people's names to application-specific addresses, and route communications between people (which we refer to as *person-level routing*).

There are naming schemes that assign unique identifiers to people; we call such names Personal Online IDs (POIDs). A POID maps to those ASAs through which a person is reachable. The use of POIDs is discussed in [MRS+99]. However, the design and implementation of our person-level router does not depend on a particular naming scheme.

The role of a person-level router is similar to that of an IP router: it takes communication from a variety of interfaces and directs it out one or more interfaces, based on the recipient's preferences and on characteristics of the communication itself. The closest current approximation is a human assistant who answers Dan's phone, reads his email and forwards his messages by calling, emailing, or paging him. Aside from wasting the assistant's time, the human approach would have difficulty handling real-time communication (e.g., forwarding an IP telephony call to Dan's cell phone).

The *Personal Proxy* (see Figure 2) is our implementation of a person-level router. It performs three distinct tasks: tracking, converting, and forwarding. As a *tracker*, Dan's Personal Proxy maintains his current accessibility information (see Section 3.2). As a *converter*, the Proxy converts communication into a form that can reach Dan, regardless of how he is currently reachable (see Section 3.4). As a *forwarder*, Dan's Personal Proxy ensures that Dan is reachable only in the ways he wishes (see Section 3.3).

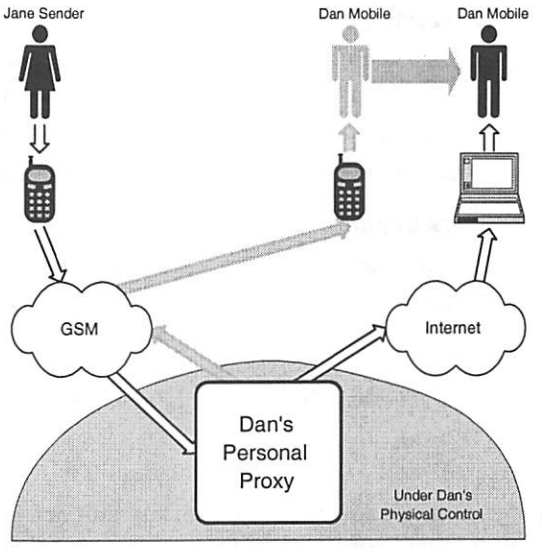


Figure 2: The Personal Proxy acts as Dan's online personal assistant. It forwards Jane's phone call to Dan's cell phone, while he is using it. When Dan stops using his cell phone and starts reading email on his laptop, the Proxy converts Jane's next call to an email message and forwards it on to the email address through which Dan is now reachable.

3 Design and Implementation

The Personal Proxy is the online analog of a mobile person's human assistant. It has its own set of application-specific addresses (ASAs), which the mobile person distributes as his own. Correspondents use these ASAs when they wish to contact him. Because the ASAs that the mobile person is really using are never revealed, his location privacy is preserved.

The Personal Proxy requires three types of information to determine how to route communication to a mobile person: the current applications through which he is available, his preferences in the form of rules, and the types of conversions the Proxy can perform. This information is used by the *Tracking Agent*, the *Rules Engine*, and the *Dispatcher*, respectively, and configured through a *User Interface*. In this section we present our extensibility model and describe each of these components.

3.1 Application Drivers

Application Drivers form the basis of our extensibility model. They are interchangeable components of the system, supporting specific programming interfaces for tasks such as protocol parsing, data transport, and filtering. Application-specific implementations of these programming interfaces can be

Dispatcher	
Session	Input Session drivers receive incoming communication. Output Session drivers generate and send out communication in application-specific formats.
Protocol	Parses metadata and content from application-specific communication formats.
Conversion	Converts data from one content type to another.
Rules Engine	
Condition	Checks a communication for particular properties.
Action	Modifies a communication in a particular way.

Figure 3: The basic programming interfaces for Application Drivers, defined for each component of the Personal Proxy.

plugged into the system at runtime and are immediately operational. Application Drivers follow the traditional object-oriented model; abstract components are created for each programming interface, which are then subtyped for specialization. Figure 3 briefly describes the driver interfaces, their functions, and the components in which they reside. The following sections will present how Application Drivers are used in more detail.

3.2 Tracking Agent

The Tracking Agent monitors the mobile person's *connectivity state*. This is a list of applications through which the mobile person can currently receive communications. For each receiving application, the Tracking Agent keeps an ASA, a protocol type and a list of communication formats, such as HTML text or JPEG image, that can be handled by the application at that ASA.

To track a user's location accurately, the Tracking Agent supports three registration methods: *scheduled*, *manual*, and *automatic*. The scheduled registration method simply assumes a change in the user's location according to a preset schedule. Manual registration requires that the user manually send a registration message to the Personal Proxy by, for example, filling out a secure web form, calling the Personal Proxy's phone number, or sending it registration email. Automatic registration is more convenient for the user, but is feasible only if the device is able to determine the user's availability and send a registration message automatically. Whatever the registration method, all registrations must be authenticated and encrypted to preserve the user's location privacy and to prevent false registrations.

3.2.1 Registration API

The Tracking Agent requires all registration clients that register on behalf of the user to conform to a specific registration Application Programming Interface. The main types of functionality included in this API are as follows:

Introspection A registration client must be able to inform the user what kinds of communication protocols (e.g., email, telephony, fax), content types (e.g., GIF89 image, RIFF audio), devices (e.g., Nokia 6190, PalmPilot V) the Personal Proxy knows about.

Customization A registration client must be able to make it easy for the user to create an initial, custom environment that can be reused in the future with minimal tweaking. More specifically, frequently-used *endpoints*, i.e., combinations of devices, protocols and content types, should be easy to alias and reuse. Similarly, frequently-used ASAs (e.g., the mobile person's work phone or instant messaging ID) or POIDs (e.g., mom's, or an employer's) should be easy to alias.

Activation Given the two functions above, a registration client must be able to mark a nicknamed endpoint as active or inactive. It should also be possible to create and activate a single-use endpoint, without having to define an alias for it first.

Our prototype currently implements two forms of manual registration: the user either indicates his connectivity state by accessing a secure web page

```
Rule #1:
  IF      From = "mom@home.net" AND
          (content contains "emergency" OR content.audio audio.pitch-is "high")
  THEN    content.text truncate-to "5 KB" AND send-to "cell phone 1"
  THEN    send-to "work email"

Rule #2:
  IF      content contains "make" AND
          content contains "money" AND
          content contains "fast"
  THEN    drop
```

Figure 4: A symbolic example of some rules accepted by the Rules Engine. Italics designate Condition or Action Drivers. *Contains* is a polymorphic condition (it can refer to text, audio, image, etc.). *Pitch-is* is a type-specific condition, as it refers only to audio content; it determines whether the pitch of the voice is relatively high. *Truncate-to* causes a content to be truncated. *Send-to* forwards a communication to the named aliased endpoint. *Drop* causes a communication to be discarded.

and downloading an applet that makes authenticated Java RMI calls to the Tracking Agent through the registration API, or sends signed email with his connectivity state. We plan to add other registration methods, including an automatic method where the user's location is tracked via a "smart badge."

3.3 Rules Engine

The Rules Engine uses the current connectivity state and the user's preferences to direct the routing decisions made in the Personal Proxy. The user enters routing preferences through the User Interface as an ordered list of *rules*. We first describe the structure of a rule. We then describe how the Rules Engine creates a set of *directives* that the Dispatcher will use when routing communication.

3.3.1 Rule Structure

Rules in the Personal Proxy follow the form

```
IF condition   THEN action
               THEN action ...
```

(see Figure 4 for an example).

Conditions are *generic*, *type-specific* or *polymorphic* and can be simple or composite (i.e., simple conditions combined using the logical connectives AND, OR, NOT). A generic condition is defined on the metadata of a communication. It is formed as a logical predicate on properties shared by all communication protocols and content types, such as size, sender or send date.

A type-specific condition is defined on one particular content type or communication protocol. For instance, such a condition could be placed on particular headers of an RFC822 email message (such as the X-Face header, which carries a low-resolution picture of the sender's likeness), or properties of an image (such as its color depth).

A polymorphic condition can be defined on many content types but has a different implementation for each one of them. Containment or similarity conditions fall within this category. Some examples are "Does the communication contain my company's logo?" or "Does the communication contain the words 'make,' 'money,' and 'fast'?" Containment in the former case invokes an image pattern-matcher whereas in the latter case it invokes a text matcher.

Actions serve two functions: *routing* and *content conversion*. Routing actions change the way a communication is delivered (or not) to the mobile person. For example, a simple one would be forwarding to a different person and a complex one would be prioritized forwarding requiring explicit acknowledgements. Content actions modify the data or convert the format of a communication. They try to massage contents to fit a particular format or set

of requirements specified by the user or obtained from the properties of the receiving application. Examples include truncating a large email sent to a pager or reducing the resolution of an image sent to a PDA.

There are two ways actions can be combined within a rule. First, actions can be composed, as is the case with the first THEN clause in Rule #1 of Figure 4. Here, the truncation occurs first and its result is sent to the application represented by the nickname "cell phone 1". Second, actions can be juxtaposed independently. In the same rule, the two THEN clauses are applied independently, in parallel or serially; the side effects of the former do not affect the latter and vice versa.

Type-specific and polymorphic conditions, as well as content actions can be installed and removed at runtime (see Section 3.1).

3.3.2 Creating Directives

When a communication arrives at the Personal Proxy, the Rules Engine determines where the communication will be routed. It evaluates the generic conditions of each rule one at a time. Conditions dependent on the type of the content (i.e., type-specific or polymorphic conditions) are evaluated as part of the conversion planning process (see Section 3.4.1). When the Rules Engine encounters a rule whose generic conditions evaluate to *true*, it assembles a set of directives derived from the remaining conditions as well as the actions of the rule. A directive consists of a *destination* and a *goal state*. A destination is a set of ASAs where the communication should be routed. This set is determined by the routing actions of the rule. The goal state contains certain requirements the Dispatcher must fulfill when handling a communication:

1. the required output content type,
2. the as yet unresolved conditions of the rule that must evaluate to *true* before the communication is routed, and
3. the content actions that are to be applied to the communication before it is routed.

Independent actions of a rule result in multiple directives. For example, Rule #1 in Figure 4 would result in two directives, one corresponding to each THEN clause. Conflicting goal states are resolved at the Dispatcher.

3.4 Dispatcher

The Dispatcher routes incoming communications to one or more ASAs, possibly converting from one application type to another along the way.

The typical path of a communication through the Dispatcher is the following (see Figure 5):

- An Input Session Driver (see Figure 3) receives the incoming communication.
- A Protocol Driver parses the communication into its metadata and its content.
- The Dispatcher queries the Rules Engine for the directives that pertain to this communication.
- The Conversion Planner (see Section 3.4.1) constructs a path through Conversion, Condition and Action drivers to bring the communication into the desired format.
- An Output Session Driver sends out the resulting communication to its destination.

3.4.1 Conversion Planning

The heart of the Dispatcher is the Conversion Planner, which transforms incoming communication from the sender into a form understandable by the mobile person's current applications or devices. The Conversion Planner has two tasks. First, it must plan and invoke a sequence of conversion drivers that implement specific transformations on the incoming communication. Examples of transformations are converting from one data type to another (for instance, text to an audio format through speech synthesis) or reducing the size of the communication (for instance, cutting all but the first 100 bytes of text, reducing the sampling frequency of a sound, or increasing the compression levels of an image). Some of the transformations might be inherently required by the mobile person's device or application (for example, the available display might have a limited color depth), some are used in a preventive manner (for example, to avoid overloading

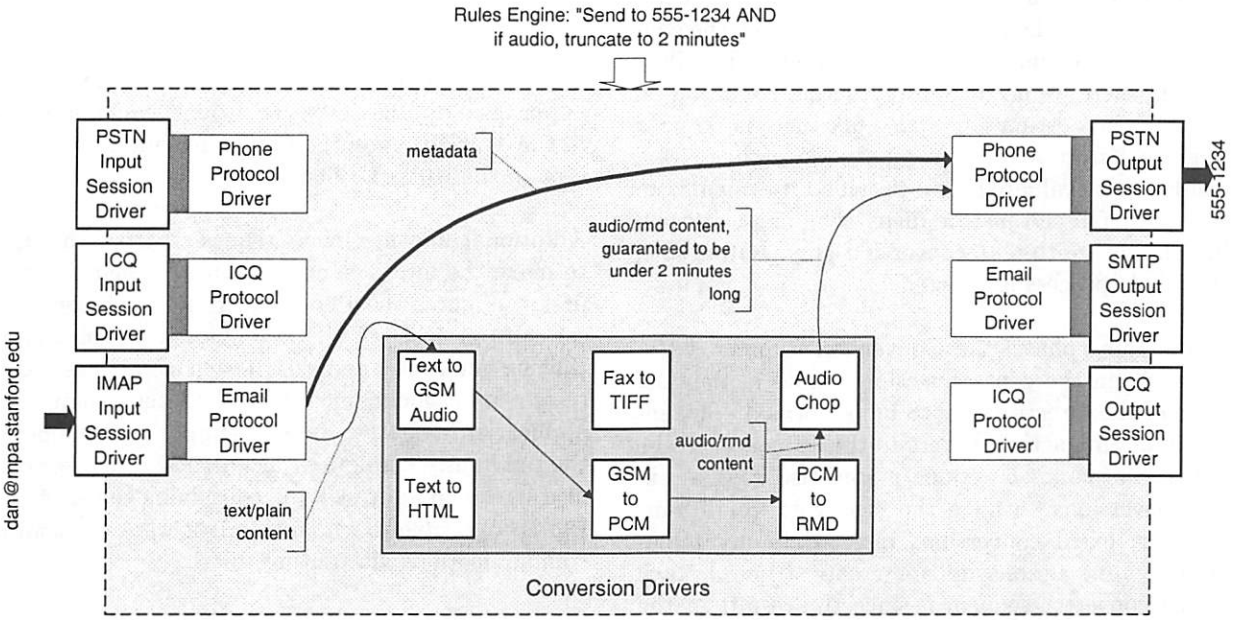


Figure 5: The Dispatcher. This figure shows how the Dispatcher routes email arriving at the Personal Proxy's email address to the telephone number at which Dan can be presently reached. Thin light arrows indicate content flow. Thin bold arrows indicate metadata flow.

the network connection or device memory), while others just aim to summarize unimportant communication selectively (for example, remove decorations from an incoming fax transmission, only leaving the text behind).

Second, the Conversion Planner must deal with type-specific and polymorphic conditions as well as content actions imposed on a communication by the Rules Engine. Such conditions or actions cannot always be applied by the Rules Engine because they might require a conversion first. For example, consider a scenario where the condition is "contains the word 'emergency'" and the incoming communication is fax. To evaluate this polymorphic condition when it has only been defined on text (i.e., when there is only a textual pattern-matcher available), the Rules Engine must invoke an optical character recognition utility to perform an image-to-text conversion, and then use the textual pattern-matcher to check for the word "emergency." To perform the appropriate invocations, the Rules Engine would in general need to implement its own planner. Rather than duplicating the planning work, the Rules Engine simply passes along to the Dispatcher those type-specific conditions, polymorphic conditions and content actions, that require some conversion before they can be used.

For each directive obtained from the Rules Engine, the Conversion Planner constructs a separate plan in three phases. In the first phase, the Conversion Planner constructs a sequence of conversion drivers. This is done through a simple breadth-first search of the conversion driver space. The end result is a sequence whose starting point is the incoming content type and whose endpoint is the required output content type listed in the goal state of the directive.

Conversion drivers are each given equal weights currently, although it would make sense to favor "cheaper" conversion drivers over more "expensive" ones, for whichever cost metric the user finds important. For example, postscript-to-text conversions require far less computation than text-to-speech conversions and if cycles are at a premium, we might want to favor the former over the latter.

In the second phase, the Conversion Planner determines when type-specific and polymorphic conditions (listed in the goal state of the directive) are to be evaluated in the sequence. If a condition cannot be evaluated anywhere in the sequence of conversion drivers, then the Conversion Planner inserts additional conversion steps to the sequence of drivers to ensure that the condition is evaluated. This might create a new branch in the sequence of drivers, since

the conversion path to check a condition might not be the same as the path needed to dispatch the content to its destination. In the example described above, where we need to use a textual condition on a fax, if the destination is also fax there is no need to go back from text to image; as soon as the textual condition is evaluated (and provided it evaluates to *true*), the fax can just be dispatched as is, discarding the image-to-text conversion step. Otherwise, the whole process is aborted.

In the third phase, the Conversion Planner determines when the content actions (listed in the goal state of the directive) are to be performed. This involves adding action drivers to the sequence. To be applicable, content actions might also require extra conversions. Unlike the case with conditions, however, extra conversions inserted to accommodate content actions do not create a new branch in the converter sequence, since the results of the content actions must be included in the outgoing content. Again, in the fax example above, if it is required to use a textual spell checker before the fax is forwarded, then an image-to-text converter, the spell checker, and a text-to-image converter have to be inserted in the conversion sequence.

We plan to explore the utility of associating an importance factor with each condition or action affecting the conversion path. This would allow low-priority conditions or actions to be ignored if they cannot be used without adding extra conversion steps.

3.5 User Interface

Unlike a network-layer router, which is usually maintained behind the scenes by a system administrator, the Personal Proxy must be configured with personal information specific to its user most likely *by* its user, to route communications effectively. Since no single user interface can work on every device from which the user might want to configure the Personal Proxy, multiple user interfaces are supported via the Tracking Agent and Rules Engine APIs. In this section we describe the design principles underlying our development of the Personal Proxy user interfaces.

3.5.1 Functionality

Two key sets of information are required of the user: application and device registrations, which are sent to the Tracking Agent, and routing rules, which are sent to the Rules Engine.

Additional information is requested from the user to make the interface easier to use. When the user first configures the Personal Proxy, he is asked to select from a list of all the devices or applications that he might use, and assign each of them an easy-to-remember nickname. When he later registers an application as active, he can simply choose one of the previously configured nicknames. The user can also set up an address book containing the ASAs or POIDs of people to whom he might want to forward communications via routing rules.

3.5.2 Supporting Different Interfaces

In keeping with the extensible nature of the system, the Personal Proxy supports any kind of user interface as long as it conforms to the APIs for the Tracking Agent and the Rules Engine.

The user interfaces we are currently prototyping are an interactive, Web-based interface which provides full configuration functionality, and a non-interactive, email-based interface which only allows registering previously configured applications.

3.5.3 Design Issues

User interfaces for communication filtering have been notoriously hard to design for several reasons. First, the structural information enclosed in filtering rules is complicated, sometimes consisting of multiple levels of a logical tree. Second, it is difficult to give the user an intuitive idea of what a certain set of rules will do when applied to a particular communication, since many rules might participate in the decision and some of their mandated actions might be mutually contradictory. Third, in an extensible filtering system such as the Personal Proxy, the interface must be flexible enough to accommodate rule components (conditions or actions) that might not have been defined at the time of interface design. The usual result is that user interfaces are either too complex for the uninitiated to use, or too simplis-

tic for experienced users to perform more ambitious tasks.

In the Web-based interface we are currently prototyping, we attempt to address each of these problems. To deal with the complex structure of filtering rules, the interface presents a two-pane display. For context, one pane shows the hierarchy of objects to which rules can be applied; the other contains the set of rules for the object currently in focus. To help the user anticipate the effect of a set of rules, the interface allows the user to send "preview" communications through the Personal Proxy and view the results without actually completing delivery. For real communications, the Dispatcher records the sequence of applied rules and conversions so that the user can later determine what went wrong if they are delivered improperly. Finally, we attack the extensibility problem by allowing developers of new rule components to override methods which define the visual presentation of each component.

4 System Evaluation

We evaluate our system using three criteria: preservation of privacy, extensibility, and ease of deployment.

4.1 Preservation of Privacy

A primary goal of our system is to preserve the privacy of the mobile person. While complete preservation of privacy is impossible [Lam73], our system goes much further towards this goal than current systems do (see Section 5). This is mainly accomplished in two ways: first, by hiding all locations visited by the mobile person and, second, by filtering incoming communications and routing them according to their desirability.

The only ASAs for Dan that Jane Sender knows of are those of his Personal Proxy. The Proxy can receive and forward all of his communications to his undisclosed, current ASAs. In addition, his registrations are encrypted and authenticated (see Section 3.2). The Proxy prevents Jane from determining Dan's exact network location, device or application, thus protecting his location privacy. Dan's location privacy is as secure as his Personal Proxy.

Table 1: Lines of code in selected Application Drivers.

Driver	Code	Doc	Total
Email Protocol	139	135	274
IMAP Input Session	108	119	277
SMTP Output Session	169	52	221
Phone Output Session	100	56	156
Text-to-Audio Conversion	39	6	45

The Proxy can run on a single host under the complete physical control of its user (although several users could share a Personal Proxy for lower cost). Consequently, our system is more secure than others that depend on several nodes, which are not all under the control of the user (or trusted third party), or of whose existence the user is not aware.

One unresolved issue concerns an adversary residing on the path between Dan and his Personal Proxy. In this case, encryption cannot preserve Dan's location privacy if registration flows are identifiable. So far we are considering disguising the flows or transmitting decoy registrations, but the adversary issue is known to be a difficult problem.

The Personal Proxy prevents unwanted communication by allowing the user to define content-based rules governing how unwanted or low-priority communication should be handled (see Section 3.3). Since rules can handle any protocols or content types, they can filter the user's communications more comprehensively than traditional filtering systems.

4.2 Extensibility

As explained in Section 3.1, extending support to a new messaging or transport protocol, or even a new content type, is merely a matter of writing appropriate new Application Drivers. The new drivers can be introduced to the running system, without requiring a restart.

We measured the extensibility of our system as the amount of programmer time and lines of code required to add functionality. Given a preexisting unofficial Java API for ICQ transport and messaging, an experienced programmer was able to create a working set of drivers for input/output sessions

and messaging in less than an hour. (ICQ [ICQ] is an instant messaging program.)

The code required to glue various other functionality into the architecture is given in Table 1. The number of lines of code and the totals for code and comments measure approximately the difficulty of adding new functionality to the architecture.

Although this gives only a rough measure of the extensibility of the system, it is safe to say that extending the system does not require bringing the system down and can be done easily, using off-the-shelf software and hardware components. It is straightforward enough that an advanced user can add functionality without days of development.

4.3 Ease of Deployment

The most important metric of a communication system is the number of people who can use it. The easier a communications system is to deploy, the more people can benefit from using it. In addition, given the rate of development of new networking technologies, a system that is difficult to deploy may become obsolete before it is widely available. The benefit of such a system is limited. For example, ISDN was not widely available in the United States until fourteen years after it was first specified [Sta94] because it required upgrading phone switches and wiring. By that time, faster technologies like cable modems and DSL were already emerging. In contrast, the World Wide Web was deployed widely in less than four years, in part because an individual can set up a web server without modifying the network infrastructure.

We have designed the Personal Proxy to be as easy to deploy as a web service. It is a single Java server that can be installed by an individual on any Java-compatible host with no special support from the underlying network infrastructure. The only requirement is that the Proxy be able to communicate with each device through which the mobile person expects to be reachable. In general, if a mobile person expects to communicate through a particular network (e.g., the telephone network or the Internet), he probably already has connections to those networks in his home or at work.

5 Related Work

Many existing solutions allow user mobility among devices in a network of a single type. Examples include the GSM and UMTS [UMT] cellular telephony systems and the Personal Mobile Telecommunications option of the Japanese cellular telephony system (PDC). All use smart cards to identify the user currently using a phone. Unlike MPA, these solutions provide user mobility within only one network type. MPA provides user mobility across all network types.

The Iceberg [JBK98] project has goals similar to those of MPA, but takes a different approach. Iceberg's functionality depends heavily on a pre-existing network infrastructure involving a large number of nodes called Iceberg Access Points (IAPs). The Iceberg design calls for tracking information and user preference information to be distributed among IAPs throughout the network infrastructure. As a result, the IAPs can avoid the level of indirection and delay added by our Personal Proxy by setting up direct and possibly shorter communication paths between the sender and receiver. However, the Iceberg design also requires that IAPs be installed in each type of network supported. For PSTN (Public Switched Telephone Network) or cellular networks, this requires modifying switches or base stations. Many small service providers may have trouble convincing telephone companies to give them access to their base stations. Given the large number of international telephone standards and network operators, it may be difficult to deploy Iceberg widely. Furthermore, Iceberg requires the user to trust several entities (the IAPs) that are not under the user's (or even necessarily a trusted third party's) complete physical control.

The TOPS architecture [AGK+99] provides both host and user mobility for telephony over packet networks. It shares with MPA the notions of a person-level addressing scheme, translating online IDs into application-specific addresses, tracking the current location of users, and converting between incompatible formats. An important aspect of TOPS that we would like to incorporate into MPA in the future is its support for capability negotiation between the caller and the callee. A key difference between TOPS and MPA is that TOPS mainly targets telephony-like applications, where real-time voice and video are the predominant content types, whereas MPA targets all communications applications, synchronous or asynchronous. Also, TOPS

pushes all filtering functionality into the Directory Service. MPA requires a Directory Service only to locate a Personal Proxy; all subsequent computation, including filtering and authentication is handled by the Personal Proxy instead. We believe this is the only way to allow directory servers to be fast and efficient, without curtailing the functionality delivered to the end user. Furthermore, TOPS exposes a person's point of attachment to others. MPA presents a black-box view to callers, allowing for better protection of the mobile person's location privacy. Another key difference is that TOPS requires all end-user applications to be rewritten. MPA can be fully operational using current applications and a Personal Proxy, which makes it much more readily deployable.

The SPIN project [LRQS97] at the Canadian National Research Council has designed a seamless messaging system whose goal is to intercept, filter, convert and deliver "multi-modal" messages including voice, fax, and email messages. Like MPA, the system performs tracking to determine the availability of the user and places emphasis on maintaining application-independent filtering rules. However the SPIN project does not try to preserve a user's location privacy. Instead this information is made globally available throughout the system. SPIN also assumes that for every data format, there will be a converter that transforms the format into a standard text format. The filtering rules are then applied to the standard format. While this eliminates the need for path planning, it introduces two problems. First, there are some data formats such as images that cannot be converted to the standard text format. Second, adding a new data format requires writing a new converter, because existing off-the-shelf converters are not likely to transform their input into the standard SPIN format. In MPA, we leverage off of existing converters and simply write wrappers around them to integrate them into the Conversion Planner.

Transcoding proxies have been researched extensively. In work by Fox et al. [FGBA96, FGCB97] the goal is to accommodate clients with limited resources and provide ways to make transcoding more scalable. Our transcoding approach does not try to improve on the results above; instead, we build smart transcoding paths, and allow the transcoding process to be coordinated with filtering and prioritization.

Many applications exist which allow the user to define rules to filter and categorize electronic mail based on metadata and keywords. This idea was explored early on in the Information Lens, a research system whose goal was to facilitate information sharing within organizations [MGT86]. Although the system allowed people to compose complex rules for messages written using semistructured templates, most people created fairly simple rules for tasks such as processing messages from distribution lists [MMC+89]. Since the Personal Proxy has no control over application-specific communication formats, it must rely on the structure provided by each application. However, the Proxy can perform conversions and route between different communication applications as well as filter and categorize, so it can provide users with a much richer set of tools for managing communications.

6 Conclusions

Ubiquitous network connectivity for devices does not automatically imply continuous reachability for people. Whereas existing mechanisms have addressed host mobility or the mobility of people within one network, few have allowed *people*, the ultimate and most important endpoints of communication, to roam freely, without being constrained to one location, one application, one device, or one network.

We propose the Mobile People Architecture (MPA) to maintain person-to-person reachability. MPA distinguishes itself from similar systems by its emphasis on protecting the user's privacy, being extensible to new network devices and applications, and being easy to deploy.

Our prototype person-level router, the Personal Proxy, currently interoperates with telephony, email, and ICQ (an instant messaging program). It filters out spam using a powerful rule-driven engine based on the mobile person's preferences. We show that the Personal Proxy restricts the trusted computing base to a single host under the physical control of the user. In addition, we show that given a library to interoperate with a new network device or application, the Personal Proxy can be extended in less than 200 lines of Java code. Finally, we show that a Personal Proxy can be deployed by an individual and used by all of that individual's correspondents without the need to modify the applications and devices they use.

The latest updates on the Mobile People Architecture can be found at http://mpa.stanford.edu/.

7 Acknowledgements

We would like to thank Ichiro Okajima of NTT DoCoMo for providing us with valuable questions and suggestions during our MPA discussions.

This research has been supported by a gift from NTT Mobile Communications Network, Inc. (NTT DoCoMo), a grant from the Keio Research Institute at SFC, Keio University and the Information-technology Promotion Agency in Japan, and a grant from the Okawa Foundation.

References

[AGK+99] N. Anerousis, R. Gopalakrishnan, C.R. Kalmanek, A.E. Kaplan, W.T. Marshall, P.P. Mishra, P.Z. Onufryk, K.K. Ramakrishnan, and C.J. Sreenan. TOPS: An Architecture for Telephony Over Packet Networks. *IEEE Journal of Selected Areas in Communications*, 17(1), January 1999.

[FGBA96] Armando Fox, Steven D. Gribble, Eric A. Brewer, and Elan Amir. Adapting to Network and Client Variability via On-Demand Dynamic Distillation. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 160–170, Cambridge, MA, October 1996.

[FGCB97] Armando Fox, Steven D. Gribble, Yatin Chawathe, and Eric A. Brewer. Cluster-Based Scalable Network Services. In *Proceedings of the Sixteenth Symposium on Operating Systems Principles*, pages 78–91, Saint Malo, France, October 1997.

[ICQ] ICQ, Inc. http://www.icq.com/.

[JBK98] A. D. Joseph, B. R. Badrinath, and R. H. Katz. The Case for Services over Cascaded Networks. In *Proceedings of WOMMOM '98*, October 1998.

[Lam73] Butler W. Lampson. A Note on the Confinement Problem. *Communications of the ACM*, 16(10):613–615, October 1973.

[LRQS97] Ramiro Liscano, Impey Roger, Yu Qinxin, and Abu-Hakima Suhayya. Integrating Multi-Modal Messages across Heterogeneous Network. In *Proceedings of the IEEE International Conference on Communications*, June 1997.

[MGT86] T. W. Malone, K. R. Grant, and F. A. Turbak. The Information Lens: An Intelligent System for Information Sharing in Organizations. In *Proceedings of CHI '86*, 1986.

[MMC+89] W. E. Mackay, T. W. Malone, K. Crowston, R. Rao, D. Rosenblitt, and S. Card. How Do Experienced Information Lens Users Use Rules? In *Proceedings of CHI '89*, 1989.

[MRS+99] Petros Maniatis, Mema Roussopoulos, Ed Swierk, Kevin Lai, Guido Appenzeller, Xinhua Zhao, and Mary Baker. The Mobile People Architecture. In *ACM Mobile Computing and Communications Review*, July 1999. To appear.

[Sta94] William Stallings. *Data and Computer Communications*. Macmillan Publishing Company, 4th edition, 1994.

[UMT] The Universal Mobile Telecommunications System. http://www.umts-forum.org/.

[WKH97] M. Wahl, S. Kille, and T. Howes. Lightweight Directory Access Protocol (v3). RFC 2251, 1997.

A User's and Programmer's View of the New JavaScript Security Model

Vinod Anupam David M. Kristol Alain Mayer
Bell Laboratories, Lucent Technologies
600 Mountain Avenue
Murray Hill, NJ 07974
{*anupam,dmk,alain*}*@bell-labs.com*

Abstract

In this paper we introduce a new security model for JavaScript in Mozilla, as well as its programming interface. We present important concepts via examples from electronic commerce applications. We also describe our experience of implementing the model in the publicly available Mozilla source code. This model is likely to be integrated into Navigator 5.0, which, as of this writing, is scheduled to be released in late fall, 1999.

1 Introduction

Web browser scripting languages are lightweight, yet powerful procedural languages with rudimentary object-oriented capabilities. Their source code is typically embedded in an HTML page and executed by an interpreter in the browser. As a form of executable content, they add interactivity and automation to browsers. This means that a Web page need no longer be static HTML, but can include transportable programs that interact with the user, control the browser, and dynamically create HTML content. Examples of such languages are Netscape's JavaScript, and Microsoft's VBScript (see [F97] and [L97], respectively).

At the same time, scripting also adds to the power of an adversarial entity. A user might visit a dubious site (`crook.com`) and (unknowingly) download scripts. Indeed, in the summer of 1997 we reported in [CERT97, AM98] an attack against both JavaScript and VBScript that allows a hostile entity to plant a Trojan Horse script in a user's browser. This script subsequently reports back all Web activity – URLs visited, private data supplied by the user in a Web form, *e.g.*, credit card numbers, social security numbers, company passwords, *etc.* Such an attack works even when the user employs encryption (*e.g.*, SSL) or when a user is behind a firewall, because the data is captured from the browser inside the firewall, before it is encrypted. We further learned that other people had discovered a series of security flaws in earlier browser versions (see [L96] for an overview). Unfortunately, the "tradition" of security weaknesses being discovered did not stop with us (see [K98] for an overview).

In March of 1998, Netscape decided to make the Navigator source code available to the public (under the name "Mozilla"). We consequently decided to implement a new security model for JavaScript in Mozilla. A technical description of the security primitives used in our model can be found in [AM98b].

We gave a demo of an early prototype "Bell Labs Mozilla" to Netscape in late fall of 1998. Given the positive feedback, we then went on to complete "our" Mozilla and ship it to Netscape in March 1999. As of this writing, our contacts at Netscape started to integrate our code into Navigator 5.0, currently scheduled for release in late fall, 1999. Check `www.mozilla.org` for updates on the progress of Mozilla.

In this paper we focus on how our new model benefits both the end user who surfs the Web and the JavaScript programmer who designs the Web sites that the end user visits. We also devote some of the discussion on our implementation experience integrating our model into the existing Mozilla source code base.

2 Brief Introduction to JavaScript

In this section, we give a very brief introduction to JavaScript; for more details see [F97, KK97]). *JavaScript* is a simple procedural language that

is interpreted by Web browsers from Netscape. (*JScript*, Microsoft's implementation, is a clone that is interpreted in Microsoft's Web browsers.) JavaScript is object-based in the sense that it uses built-in and user-defined extensible objects, but there are no classes or inheritance. The code is integrated with, and embedded in, HTML. By default, JavaScript provides an object-instance hierarchy that models the browser window and some browser state information. For example, the *navigator* object provides information about the browser to a script, and the *history* object represents the browsing history in the browser window.

Also, through a process called *reflection*, JavaScript automatically creates an object-instance hierarchy of elements of the script's HTML document when it is loaded by the browser. The *location* object represents the URL of the current document, while the *document* object encapsulates HTML elements (forms, links, anchors, images, *etc.*) of the current document. The reflection process defines a unique *name space* for each HTML page and thus for each collection of scripts embedded in that page. JavaScript is *loosely typed*: variables' data types are not declared. JavaScript uses *dynamic binding*: object references are resolved at runtime.

3 Overview of our Security Model

In this section, we give an overview of the security primitives used in our model; for more details see [AM98b]. Our design is based on the following two basic building blocks: (1) *Access Control* regulates what data a script can access on a user's machine and in what mode; and, (2) *Trust Management* regulates how trust is established and terminated among scripts executing simultaneously in different contexts. See [AM98b] for a detailed discussion and justification of the security primitives employed and what common goal is being realized through them.

3.1 Security Policy and Access Control

Our JavaScript interpreter accepts as input, in addition to the scripts to execute, a *security policy* from the browser user. Different users may have different requirements with respect to their own privacy or that of the data they submit, and this will be reflected in their chosen security policies. Simply put, a security policy defines a partitioning of the JavaScript name space into inaccessible, read-only, and read-write objects.

A policy also defines the action for the JavaScript interpreter to take when the current script tries to execute an operation that violates the access control specification that the policy's name space partition defines.

A security policy further specifies which external protocols (*e.g.*, loading a `mailto:` or `ftp:` URL) the script is allowed to invoke and the appropriate action to take in the event that a script attempts to invoke a protocol not allowed by the policy.

3.2 Management of Trust

We use *access control lists* (ACLs) to regulate access that scripts have to objects in name spaces other than their own (*e.g.*, a page in a different browser window or frame). In a nutshell, a document's ACL is a list of URL paths or hostnames. Only a script whose origin appears in the document's ACL may access the name space of this page. The ACL mechanism allows Web developers to both expand and contract the set of domains that they trust. For example, `store.com` can put `partner.com` on the ACL of an HTML document that it serves to allow scripts from `partner.com` full access to the page. Also, `emall.com/store1` can prevent scripts from `emall.com/store2` from accessing its documents by setting its ACL to `emall.com/store1`.

The ACL provides an all-or-nothing control for access to a name space by other scripts. Another script is *trusted* by a document if the script's origin is listed in the document's ACL. Either every object in the document's local name space is accessible (to a trusted script), or none is (to an untrusted script). To complement this, we introduce a new method, `setPrivate`. If for any object `obj` a script executes `setPrivate(obj);`, then `obj` (and any of its properties) is subsequently inaccessible, even to trusted scripts.

4 The End User's View

In current browsers, the user's choice with respect to JavaScript is truly limited. He/she can either turn JavaScript completely off or on for all sites.

We introduce the notion of a *security policy* for JavaScript. From the user's perspective, a security policy is a bundled set of preferences with respect to the following capabilities given to scripts that execute on the user's machine:

- *Access to reflected objects*: A script has access to a number of objects in the JavaScript hierarchy. For example, `document.referrer` indicates the page from which the user arrived at the current page, and `navigator.platform` indicates the operating system on the user's machine. A policy aggregates access permissions to these *property policies* of all reflected objects.

- *Access to external interfaces*: A script also has access to a number of external interfaces, such as ftp (`ftp:` URL) and e-mail (`mailto:` URL) protocols. It also has access to calls in the Java language, through which it can capture the user's IP address, for example (by calling `java.net.InetAddress`). Again a policy aggregates access permissions to all external interfaces.

- *Actions in the event of access violations*: A policy also specifies the action to take by the JavaScript interpreter if a script attempts to violate the current policy.

Most end users will not want to be bothered with such low-level details as the exact specification of a policy. Thus we offer a small number of increasingly strict *predefined policies* from which the user can pick; see Figure 1. The chosen policy, the *global security policy*, will be in effect whenever the user starts visiting Web sites. A user can also pick predefined policies to be in effect only for specific sites (*site-specific security policy*). The user may specify either a hostname or a specific URL for which this policy should be in effect; see Figure 2. For example, it makes sense to allow a more lenient policy when browsing within an intranet than when accessing the external Internet. In fact, as part of an overall corporate security policy, the employees' browsers can be initialized with a strict policy for external sites and a liberal policy for internal sites.

As for many security tools (see, *e.g.*, [WT98, ZS96]), it is hard to design a user interface that, on the one hand, does not restrict the power user from fully exploiting the provided functionality, and, on the other hand, does not confuse the average user, the confusion leading to possible unwanted security implications. For example, [WT98] call the problem of choosing access rules and policies the *abstraction property* and observe that such notions are often alien and unintuitive to a wider user population. Another factor mentioned in [WT98] is that users get little feedback when they make an error in configuring security aspects. Consequently, we think it is very important that reasonable default settings be chosen for the average user and that good representative examples be chosen to explain in which situations a given policy is adequate.

We envision that corporate administrators will want to incorporate policy management (creating and updating policies, installing new policies on each desktop's browser, etc.) via some sort of directory service integration (e.g., LDAP-like solution). Home users might also want to have tools to easily create or download (from certifiably trusted site) and then install new policies on their desktop. Thus, policy management and its tools seem like a fruitful area of further work.

4.1 Signed JavaScript

Netscape Navigator 4 and later versions support digitally signed scripts that can request privileges, and, subject to user approval, lift certain security restrictions while executing. A digital signature allows the browser to securely establish the author of a signed JavaScript program (see [N98]). Cryptographically signed scripts are not yet very popular, partly because average users find it hard to grasp the privilege-granting process or the implications of granting a particular privilege.

For future versions of browsers we propose to integrate code signing into our model, by having specific security policies that go into effect if a signed script is downloaded from a particular site. For example, a Fidelity policy for the user's interaction with the brokerage house might allow reading and writing files in a specific directory, so that the user can study his account offline.

5 The JavaScript Programmer's View

5.1 Domain and URL Based Trust Management

JavaScript executes in the name space defined by both the browser window and the HTML page in which it is embedded. This name space is accessible to all scripts embedded in the same page. Standard JavaScript also grants access to that name space to scripts that run in other browser windows, but that were loaded by an HTML page that loaded from the same server. This *same origin policy* leads to the situation where scripts from, for example, `e-mall.com/pet-shop` can access all data (*e.g.*, credit-card numbers) that the user inputs into a form at

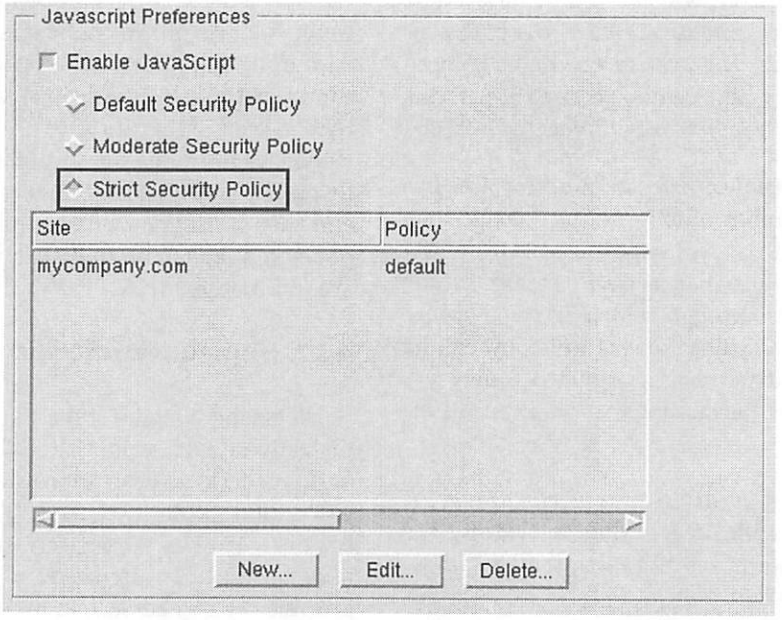


Figure 1: User Interface for Setting Browser Security Policy

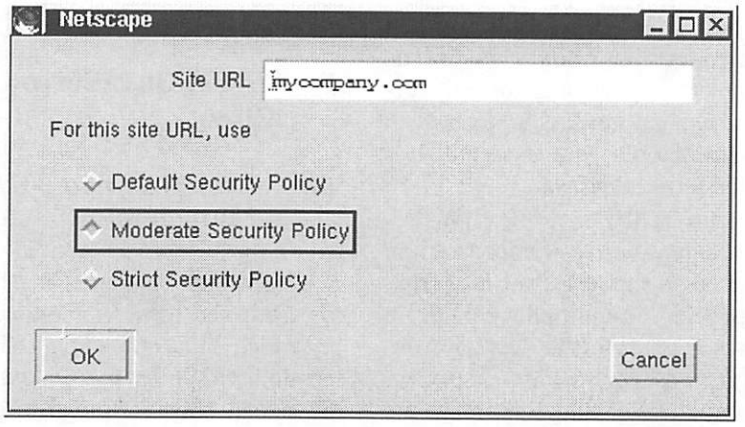


Figure 2: User Interface for Setting Site-Specific Security Policies

https://e-mall.com/toy-store/checkout.html, given that the user has pages from both shops open at the same time. Note that this access is possible even though toy-store uses SSL to secure their client's data.

In our model, the JavaScript programmer can model trust *explicitly* by using ACLs. The toy-store programmer can state in the initialization step of scripting in all of the toy-store pages:

```
<SCRIPT LANGUAGE="Javascript">
document.ACL =
  "http://e-mall.com/toy-store";
...
</SCRIPT>
```

The above statement indicates that only scripts from a URL that is prefixed by the above element of the access control list (ACL) are allowed to access the page's name space. Thus, if a script embedded in e-mall.com/pet-shop/snoop.html executes the following:

```
<SCRIPT LANGUAGE="Javascript">
toy_store_check_out_window =
  window.open("http://e-mall.com
               /toy-store/checkout.html");
...
```

the page will be loaded into a new browser window on the user's desktop, but its name space will be inaccessible to the calling script.

If toy-store decides to collaborate with baby-store in order to cross-link, then the initialization might look like:

```
document.ACL =
   "http://e-mall.com/toy-store
    http://e-mall.com/baby-store";
...
```

If these two stores want to collaborate further with a site that is not even part of the e-mall domain (*e.g.*, the parentsoup site), then while the existing browsers do not allow this, our model can accommodate this easily by using the following:

```
document.ACL =
  "http://e-mall.com/toy-store
   http://e-mall.com/baby-store
   www.parentsoup.com";
...
```

The last entry above is a domain name, which includes all pages from that domain in the ACL.

5.2 Fine-Grained Trust Management via SetPrivate

The ACL-based approach is still an all-or-nothing approach in the sense that by including a page in its ACL, a script makes its *whole* name space available to the other page. This coarse granularity may not always be appropriate. The new security model allows Web developers to prevent access to sensitive information by marking it private by using the setPrivate method. We motivate this by using an example.

"Associates programs" are rapidly gaining popularity with e-commerce sites, such as amazon.com. In their associates program, amazon.com pays each participating site (like associate.com) a small percentage of a sale that results from a person's following a link from associate.com to amazon.com, *i.e.*, by a person who was *referred by* associate.com. Often, the associates program is realized by a third-party network, such as linkexchange.com. Currently, both associate.com and linkexchange.com simply have to trust amazon.com to provide at the end of each month a statement that accurately reflects the sales; the business relationship is clearly stacked in favor of amazon.com — since linkexchange.com and amazon.com are different domains, the name spaces on their respective pages are inaccessible to each other. The same applies for associate.com and amazon.com. At best the associate could know that the user clicked on their link leading them to amazon.com by appropriately instrumenting their Web page, and the third-party network could know that this happened if referrals were redirected through it. However, there is no way for either of them to verify that the user engaged in a purchase. While this may not a problem with a well-known site such as amazon.com, better accountability would boost these programs when other stores (new-vendor.com) are involved.

While using ACLs would circumvent the problem of lack of access, new-vendor.com may not be comfortable simply setting

```
document.ACL = "www.associate.com";
```

on all of its relevant check-out pages, as that could potentially reveal confidential client data (*e.g.*, credit-card number, etc.). However, new-vendor.com might be willing to reveal some of the non-confidential customer data, such as the fact that the referred client did indeed purchase a book and the amount paid. In our model,

new-vendor.com would simply add the following code on the check-out page:

```
<SCRIPT LANGUAGE="Javascript">
document.ACL = "www.associate.com";
setPrivate(CreditCardForm, "elements");
....
</SCRIPT>
```

The above code allows access by scripts from www.associate.com to all the elements on the page except the form with the name "CreditCardForm". Assuming that all the client's confidential data is in this HTML form, it is now protected. We note that the statement setPrivate(document.forms[0], "elements"); is equivalent to the above, if "CreditCardForm" is the first form on the HTML page. But we caution that this version is less safe. If during page evolution, a new form is inserted in front of "CreditCardForm", the new form will be protected while "CreditCardForm" is all of a sudden accessible again to www.associates.com. Therefore we strongly recommend the use of names in setPrivate.

Now, when associate refers a potential client to new-vendor, its page can stay resident on the user's desktop in its own window or frame; if the client ever reaches the check-out page at new-vendor, associate will be able to read the relevant data used to verify new-vendor's end-of-month statements. Note that new-vendor.com can generate the necessary JavaScript automatically via server-side logic based on referrer information. The ability to selectively expose information realizes a better balance in the business relationship between a store and its associates. Furthermore, as soon as the user leaves new-store.com's Web site, the associate no longer has access to any information about the user.

A similar scenario exists for sites like shop.com that serve as centralized resources providing information about online stores. shop.com's customers (online vendors) provide it with enough information to organize stores into hierarchies, provide searchable interfaces, *etc.* A user browsing shop.com's Web site eventually clicks on a link on a page from shop.com and is sent to store.com's Web site, which is displayed as a frame on shop.com's page. As in the earlier example, shop.com only knows that the user clicked through to store.com, and is unaware of any activity that subsequently transpires.

The new security model allows shop.com's customers (store owners) to put shop.com on the ACL of pages they serve. store.com appropriately protects the information that it considers sensitive (*e.g.*, the user's credit card number) by marking it private. shop.com thus has access to the information that it needs for pay-per-click, pay-per-lead and pay-per-sale type scenarios.

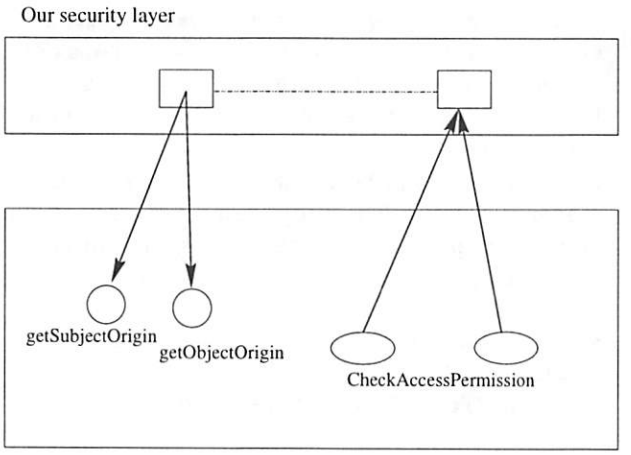


Figure 3: Security Layers

6 Security Code Layers and Hardening of the Mozilla Layer

Our implementation adds a new security layer on top of the existing Netscape code, realizing access control, security policies, and trust management as described in Section 3. We created a security layer API and added calls to it from Netscape's code, as depicted in Figure 3. (See Section 7 for further details.)

The robustness of the combined code depends on finding all the right spots in Netscape's code at which to interpose our API calls such that we close all back doors. At the same time, our implementation makes calls to basic functions in the Netscape code and therefore relies on the correct behavior of that code. Much of that code is devoted to identifying the subject and object origin URLs. (The subject origin URL is the place where the executing JavaScript code comes from. The object origin URL is the place where the JavaScript code comes from for the object being acted on.) If our code were to get the wrong information, it could possibly grant access inappropriately, thus opening a security hole. Given the importance of this basic code, we suggest a more methodical approach to realize these two basic functions. (See the subsequent subsections.)

Another area of concern is that object values persist across document loads in a window. Each document is supposed to form a separate con-

text. However, in Netscape's current implementation, `window.name` maintains its value across document loads. A clever intruder could then access the information that was supposed to be destroyed with the object. While we have fixed the case of `window.name`, we chose not to close this hole in its generality, because we believe that the new Netscape document object model (DOM) would do so for us.

6.1 Finding the Subject Origin URL

JavaScript has a number of possible ways to pass the control flow or to generate an additional thread of execution:

1. Function/method call: *e.g.*, `v = foo(x);` or `u = otherWindow.foo(x);`
2. Dynamic generation of JavaScript code: *e.g.*, `document.write(foo(x));` or `myWindow.eval(foo(x));`.
3. JavaScript URL: *e.g.*, `otherWindow.location = "javascript:foo(x)";`
4. JavaScript invocation through HTML (browser):

   ```
   <a target=otherWin href="javascript:
      alert(window.location);" >
   ```

   Also, code invoked through installed event handlers, "script" tags, *etc.*

Most "traditional" languages only have function/method calls. In those cases it is usually fairly straightforward to determine which entity originated the call sequence by inspecting the call stack, *i.e.*, stack inspection. The additional methods and flexibility in JavaScript complicate this task. For instance, dynamic invocation has the flavor of a function call, but the passing of control does not take place instantly. In fact, a callee might be executing at a time when the caller is no longer on the stack. Therefore, we propose the use of **proactive forward passing** of the subject origin information. Whenever a passing of the control flow in the code is indicated (and whether it is about to take place right then or not), the interpreter sets the subject domain of the callee to the subject domain of the caller, in such a way that when the callee executes, this value can be easily retrieved.

We distinguish three cases:

- The callee will begin executing immediately. Either its stack frame is loaded right on top of the caller's frame (*e.g.*, regular function call) or its stack frame is loaded onto another document's stack frame (*e.g.*, `javascript` URL). Hence, the interpreter can propagate the subject origin information among stack frames (which logically form a tree structure).

- The callee will execute at a later point (*e.g.*, `document.write`). In this case, the callee will be the top-level stack frame in its document. Hence, the interpreter can store the subject origin information as an attribute of the document. Once the top-level stack frame gets loaded, the interpreter fetches the subject origin information from the document attribute and initializes the subject origin value in the top-level stack frame.

- JavaScript is invoked through HTML. In this case, the callee executes in a top-level frame of the JavaScript stack of its document. Hence, the browser should play the role of a JavaScript caller as far as the passing of the subject origin to the top-level frame is concerned.

Our goal is to arrive at a situation where, for every stack frame loaded, the correct subject origin can be retrieved from a well-defined location and, consequently, there will never be the need to search backward for it or even to fail and declare "unknown origin", as might occur in the current version of JavaScript (4.x browsers).

6.2 Finding the Object Origin URL

The object origin is always defined by its static scope. That is, global objects always derive their origin from the origin of their document. Similarly, an object local to a function or method derives its origin from the origin of the method. A reference to a variable x in a document d loaded into window w is really $w.x$ and thus derives its object origin from the origin of d. Let y be a local variable to the function `foo()`, located in a document d in window w. Even when `foo()` is called from a different window `otherWin`, *e.g.*, when "`z = w.foo();`" is a line in `otherWin`, the object origin of y is still determined by its static scope, the origin of d. Note that in "`otherWin.location = "javascript: x = 1;";` ", the "`"javascript: x = 1;`" part is just a string and not code; thus the static scope of x is within the window `otherWin`.

Dynamically created documents (*e.g.*, created by `document.write`) should always inherit origin from the creating window/document.

Here are some examples for code executing in a window $w1$, loaded from origin $o1$:

```
v = foo(x);
  // o1 remains subject origin
  // for function ''foo''
v = w2.foo(x);
  // o1 remains subject origin
  // for function ''foo''
  // foo will execute in w2's scope
w2.location = "javascript:foo()";
  // o1 remains subject origin
  // for function ''foo''
  // foo will execute within a
  // top-level stack frame
  // of a new document in w2.
w1.document.write(foo());
  // o1 remains subject origin for
  // function ''foo''
  // foo will execute in a new
  // document in w1, whose origin
  // is the same as the one of
  // the current document
w2.eval(w1.foo(x));
  // subject of this statement is o1 and
  // thus is subject of foo.
  // foo will execute in w2's context,
  // since w2 owns the eval method.
```

Thus, in this context, the object origin can always be retrieved in one step, by accessing the enclosing scope.

7 Our Implementation

Our implementation adds a new security layer on top of the existing Netscape code, realizing access control, security policies, and trust management as described earlier.

7.1 Overview

With the exception of user interface code to support site security policies, nearly all the code that was modified directly supports the JavaScript object model.

Property policies are implemented in the respective modules for their objects. They control whether there is read-write, read-only, or no access to the property. External interface policies are handled by the code that sets a URL object's value. They control whether there is read-write, read-only, or no access to the external interface.

Our implementation depends on correctly identifying subject and object origin URLs. The subject origin URL determines which site security policy to use. The subject and object origin URLs together determine ACL behavior.

When a policy violation of any kind occurs, the implementation always presents an error dialog to the user. Based on the value of a configurable continuation preference setting in the current policy, the JavaScript interpreter may then stop interpreting the offending script, or it may continue, while denying the requested access.

7.2 Policy Lookup

To look up a policy P for a property or external interface, our implementation first checks whether there is a site security policy for the executing script's origin URL. Otherwise it uses the current default global security policy. It then checks whether there is a preference for P in that security policy.

Because we allow site security policies to apply to URLs, and not just to hostnames, the site security policy lookup uses the longest (and, therefore, presumably most specific) policy that matches the subject URL. For example, assume there are site security policies for `e-mall.com` and `http://e-mall.com/store1/`. The subject origin URLs `http://e-mall.com/index.html` and `http://e-mall.com/store2/` would both use the first policy, whereas, `http://e-mall.com/store1/catalog.html` would use the second.

7.3 Property and External Interface Policies

The per-property and per-external-interface policies were easy to implement. At the point in the code where a get- or set-property function is implemented, the modified code checks whether there is a corresponding property (external interface) policy in effect, and, if so, whether the requested access violates it (for example, attempting to write a read-only property). Because this check happens frequently, performance optimizations should be considered, such as caching previous results of checks or building a hash table that can be efficiently queried to check whether an object's access is affected by an existing policy. A vast majority of objects won't

be affected since a typical policy covers only a few security-sensitive objects.

On a violation, the implementation checks the continuation setting for the relevant security policy and either aborts interpretation or continues without granting access. If there is no violation, interpretation proceeds normally.

Access to the new `document.ACL` property is a special case. We unconditionally restrict access so only the script that created the `document` has permission to read or write `document.ACL`. Otherwise a rogue script could attempt to change `document.ACL` and gain access to the objects that ACLs protect.

7.4 Access Control List (ACL) Support

Because an ACL is tied to a `document` object, the default ACL is the subject origin URL that created the document. However, as stated earlier, that script may change the value of `document.ACL` to enlarge the set of URLs that can access objects that the ACL protects.

We implement ACL checks in those places in the interpreter code where one object attempts to access an object that might have arisen from a different context. The implementation checks the ACL for the (containing object of the) object being accessed against the accessing script's (subject) origin URL. If the two are identical, or if one of the URLs in the ACL is a prefix of the subject origin URL, then the check succeeds, and access is granted. Otherwise an ACL violation occurs. As above, the continuation policy on ACL violation is controlled by a preference, but access to the protected object is never granted if a violation occurs.

As described above, our current implementation uses only prefix matching for ACL checks rather than fully general regular expressions, which are costly to evaluate. Logically, an ACL check has to be performed for each object access. Note however, that the ACL check actually has to be performed only once for each ACL/subject pair if we cache the outcome of the check (and adjust or flush the cache when there is an assignment to an ACL). Thus under such cache optimization, the use of regular expressions becomes feasible.

7.5 setPrivate, unsetPrivate

Even if access is granted according to the ACL, the specific object may have been `setPrivate`. The implementation does a separate check for whether the object is `setPrivate`, and there is a separate continuation policy for failed attempts to access one.

As with `document.ACL`, we only allow the script that created a document to `set` or `unsetPrivate` objects.

7.6 Hierarchy of Tests: "Hedging Your Bets"

The various tests outlined above and summarized here must *all* succeed before access to an object or property is granted. The order of checks is something like this:

1. Check whether the (JavaScript) script is signed, and if so, whether the signature is valid. (The current Netscape security model does not allow access to some objects/methods unless the script presents a valid signature. One example is the user preferences object, `navigator.preferences`. We integrate this approach into our access control.)
2. Check for ACL violation.
   - Determine whether subject has permission to access object.
   - Check that the accessed object has not been `setPrivate`
3. Check for property policy violation
   - Determine which security policy applies (in order): site policy, global policy, default policy.
   - Check whether there's a property policy under the applicable security policy.
   - Check whether the property policy has been violated

Note that this hierarchy means that even a signed script is not granted unconditional access to JavaScript objects. A signed script makes some parts of the object model accessible that otherwise would not be, but the signed script's code is still subject to the same set of checks as any other script.

7.7 User Interface Changes

We pre-defined three model security policies: `strict`, `moderate`, and `default`, where `default` corresponds to Mozilla's behavior prior to our changes. When a user enables JavaScript in Advanced Preferences, she can now select one of these three security policies as her global security policy. Moreover, there is a table of site security policies, initially empty, where she can add, delete, or edit

site security policy selections. This facility allows her to enter a hostname or URL in a text field and to mark a checkbox for the desired policy (`strict`, `moderate`, `default`).

8 Summary and Outlook

We have described the benefits of our model and implementation for the end user and the JavaScript programmer. Our model furthermore benefits the Mozilla developers: If new security bugs emerge, browser developers can use specialized fine-grained policies that control access to sensitive objects in order to identify paths that need to be followed to mount a successful security breach. Furthermore, in case of such a security bug, Mozilla can make available specially tailored policies to download, which protect against exploitation of this security bug. This is a more desirable course of action than the current practice of suggesting turning off JavaScript entirely until a bug fix is available. (Our current implementation allows only predefined security policies, but it could be extended to provide for user-defined policies.)

Our positive experience with using site-specific security policies indicates that such policies for the whole browser (on-off for cookies, Java, JavaScript, *etc.*) should be considered.

We hope that the final implementation will be scrutinized early on by the Mozilla open source community, so that remaining weaknesses can be identified before they become actual "bugs".

Acknowledgments: We thank Murali Rangarajan, who did the initial work on the implementation of the new security model. We are grateful to Norris Boyd and Tom Pixley at Netscape for their help and encouragement. Finally, Eric Brewer at Inktomi, in his role as USITS shepherd, helped us to make a number of improvements.

References

[AM98] V. Anupam and A. Mayer, *Security of Web Browser Scripting Languages: Vulnerabilities, Attacks and Remedies*, Proc. 7th USENIX Security Symposium, January 1998.

[AM98b] Vinod Anupam and Alain Mayer, *Secure Web Scripting, IEEE Internet Computing*, Nov/Dec 1998.

[CERT97] CERT* Advisory CA-97.20, *JavaScript Vulnerability*, CERT Coordination Center, July 1997, `ftp://info.cert.org/pub/cert_advisories/CA-97.20.javascript`.

[F97] D. FLANAGAN, JavaScript: The Definitive Guide. *O'Reilly and Associates*, January 1997.

[KK97] P. KENT, J. KENT, Official Netscape JavaScript 1.2 Book *Netscape Press & Ventana.*

[K98] E. KUBAITIS, WWW Browser Security and Privacy Flaws, `http://www.cen.uiuc.edu/~ejk/browser-security.html`

[L96] J. R. LoVerso, *JavaScript Security Flaws*, `http://www.schooner.com/~loverso/javascript/`

[L97] P. Lomax, *Learning VBScript*, O'Reilly and Associates, July 1997.

[N98] Netscape Corp., *JavaScript Security in Communicator 4.x*, `http://developer.netscape.com/docs/manuals/communicator/jssec/contents.htm`

[WT98] A. Whitten and D. Tygar, *Usability of Security; A Case Study*, CMU Tech. Report 98-155.

[ZS96] M. E. Zurko and R. Simon, *User-Centered Security, New Security Paradigm Workshop*, 1996.

Using Full Reference History for Efficient Document Replacement in Web Caches

Hyokyung Bahn[1] Sam H. Noh[2] Sang Lyul Min[3] Kern Koh[1]

[1]*Department of Computer Science, Seoul National University, Seoul 151-742, Korea. http://oslab.snu.ac.kr*

[2]*Department of Computer Engineering, Hong-Ik University, Seoul 121-791, Korea. http://www.cs.hongik.ac.kr/~noh*

[3]*Department of Computer Engineering, Seoul National University, Seoul 151-742, Korea. http://archi.snu.ac.kr*

Abstract

With the increase in popularity of the World Wide Web, the research community has recently seen a proliferation of Web caching algorithms. This paper presents a new such algorithm, that is efficient and robust, called Least Unified-Value (LUV). LUV evaluates a Web document based on its cost normalized by the likelihood of it being re-referenced. This results in a normalized assessment of the contribution to the value of a document, leading to a fair replacement policy. LUV can conform to arbitrary cost functions of Web documents, so it can optimize any particular performance measure of interest, such as the hit rate, the byte hit rate, or the delay-savings ratio. Unlike most existing algorithms, LUV exploits complete reference history of documents, in terms of reference frequency and recency, to estimate the likelihood of being re-referenced. Nevertheless, LUV allows for an efficient implementation in both space and time complexities. The space needed to maintain the reference history of a document is only a few bytes and furthermore, the time complexity of the algorithm is $O(\log_2 n)$, where n is the number of documents in the cache. Trace-driven simulations show that the LUV algorithm outperforms existing algorithms for various performance measures for a wide range of cache configurations.

1. Introduction

In an effort to relieve the problem of network congestion and latency on the World Wide Web (WWW), the research community has recently seen a proliferation of Web cache replacement algorithms [1, 2, 3, 5, 6, 8, 9, 10, 14]. This paper presents yet another such algorithm. However, the algorithm that we propose has the following salient features:

- First, it shows the best performance for a wide range of cache configurations in terms of popular performance measures used in evaluating Web caching algorithm, namely, the hit rate, the byte hit rate, and the delay-savings ratio.

- Second, the proposed algorithm makes full use of all of the past activities, in terms of reference frequency and recency, made upon the Web cache in deciding which document to evict. By so doing, it not only accurately distinguishes actively referenced documents and those that are not so, i.e. hot and cold documents, but it also distinguishes those documents that are hot but are getting colder, and those that are cold but are getting hotter. This is the main reason behind the superior performance.

- Third, the implementation of the replacement algorithm is efficient in both space and time complexities. The space needed to maintain the reference history of a document is only a few bytes per document and furthermore, the time complexity of the algorithm is $O(\log_2 n)$, where n is the number of documents in the cache. This is a feature that is not easy to satisfy when the *size* of the document and *cost* of fetching documents from remote sites have to be incorporated into the algorithm, as it is in Web caching [5,6,8].

- Fourth, the replacement algorithm retains the above features irrespective of what the performance measure of interest is. As mentioned above, in Web

caching environments, the typical performance measures of interest are the hit rate, the byte hit rate, and the delay-savings ratio. The proposed algorithm can easily be conformed to execute based on a particular performance measure. This is unlike many previous algorithms that tightly couple the optimization of a particular performance measure into the algorithm itself.

Before describing the algorithm, we describe, in the next section, the measures that have been the focus of interest and optimization in the Web caching realm. We also briefly make comparisons of previously presented algorithms. We focus on the differences and main features of the algorithms rather than describing them individually in detail. In Sections 3 and 4, we describe the proposed algorithm, namely, the Least Unified-Value (LUV) algorithm and its implementation. In Section 5, we describe the results of the simulation experiments, and compare LUV's performance with previously proposed algorithms. Finally, we conclude in Section 6.

2. Performance Measures and Related Works

Performance measures of interest in the Web caching realm can be defined according to the goal of caching. The three popular performance measures used in Web caching, that is, the hit rate, the byte hit rate, and the delay-savings ratio, denoted as HR, BHR, and DSR, respectively, can be described as follows:

$$\text{HR} = \sum h_i / \sum r_i$$

$$\text{BHR} = \sum (s_i \cdot h_i) / \sum (s_i \cdot r_i)$$

$$\text{DSR} = \sum (d_i \cdot h_i) / \sum (d_i \cdot r_i)$$

where

- h_i : number of hit references to document i,
- r_i : total number of references to document i (number of hits + number of misses),
- s_i : size of document i,
- d_i : delay time to fetch document i from the original server to the cache.

HR is the measure used in traditional caching systems such as file caching and database buffer management, and represents the number of hit references over the total number of references. BHR represents the number of bytes saved from retransmission by using the cache over the total amount of bytes referenced. BHR considers the size of the Web document, but does not consider the difference in retrieval costs. Among documents that are of the same size, those that incur higher cost in retrieving the document should be retained in the cache longer than those that incur lower cost. DSR, which considers this matter in terms of retrieval latency, represents the reduced latency by virtue of a cache hit over the total latency incurred when assuming caches are not used [8]. One may also define other new performance measures that reflect the focus of interest one wants to measure. For example, Kelley et al. define VHR (value hit rate) which represents a normalized measure of social welfare [14]. In this case, performance is measured by simply replacing the delay time (d_i) in DSR by *value* in VHR.

Many of the previous algorithms proposed for Web caching have attempted to optimize performance for one particular measure. The LRU, LFU, SIZE [9], HYBRID [10], and LNC-R-W3 [8] are such algorithms (Note the *Performance Measure* column of Table 1). These algorithms have a weakness that as the performance measure of interest changes due to circumstantial changes, they have difficulty in adjusting to these changes. The GD-SIZE [2], LRV [5], sw-LFU [14], and LUV algorithms, on the other hand, are robust to changes in the performance measure of interest. However, for the LRV algorithm, the time complexity varies according to the performance measures. While the algorithm is efficient for the byte hit rate measure, for other measures the complexity of the algorithm is $O(n)$, which makes it impractical as an on-line algorithm. For this reason, a recent version of LRV restricts their optimization of the algorithm to only the byte hit rate measure [15]. Another weakness of the LRV is that its replacement decision is based on extensive empirical analysis of trace data. Finally, the MIX algorithm tries to optimize the combination of the three measures, i.e., HR, BHR, and DSR [6].

For caching algorithms to be practical, it is important that the time complexity of the algorithm not be excessive, preferably not higher than $O(\log_2 n)$ where n is the number of documents in the cache [2,4]. Algorithms that do not meet this criterion are the LNC-R-W3 and MIX algorithms (Note the *Complexity* column of Table 1).

Table 1. A summary of Web caching algorithms.

Algorithm	Maintaining Reference History		Performance Measure for which the Algorithm Optimizes	Complexity (n is the number of cached objects)		Advantage/Weakness	
	Recency	Frequency		Time	Space	Advantage	Weakness
LRU	Last reference time	No	Fixed to BHR	$O(1)$	$O(n)$	Simple to implement	Fixed performance measure; Considers partial aspects of reference history
LFU	No	Number of references while in cache	Fixed to BHR	$O(\log_2 n)$	$O(n)$		
SIZE [9]	No	No	Fixed to HR	$O(\log_2 n)$	$O(n)$	Keeps many documents in cache	Fixed performance measure; Does not consider reference history
HYBRID [10]	No	Number of references while in cache	Fixed to DSR	$O(\log_2 n)$	$O(n)$ + per server information	Good estimation of download latency	Per server information overhead; Fixed performance measure
LNC-R-W3 [8]	k-th reference time	Based on k-th reference time	Fixed to DSR	$O(n)$	$O(kn)$ + replaced document's information	Perception of normalized contribution to DSR	Time complexity; Space overhead
LRV [5]	Last reference time	Number of references	Fixed to BHR	$O(1)$	$O(n)$ + replaced document's information + parameter value	Considers trace characteristics	Trace analysis overhead
			Any other measure	$O(n)$			
GD-SIZE [2]	Last reference time	No	HR, BHR, DSR, or any other measure	$O(\log_2 n)$	$O(n)$	No parameter; k-competitive; Any performance measure possible	Does not consider frequency; May cause cache pollution with high cost documents
MIX [6]	Last reference time	Number of references while in cache	HR, BHR, and DSR combined	$O(n)$	$O(n)$	Considers all primary parameters	Time complexity
sw-LFU [14]	Last reference time used only as a tie breaker	Number of references while in cache	HR, BHR, DSR, or any other measure	$O(\log_2 n)$	$O(n)$	Any performance measure possible	Does not consider recency except to break ties
LUV	Time of all past references	Number of references while in cache	HR, BHR, DSR, or any other measure	$O(\log_2 n)$	$O(n)$	Uses complete reference history; Best performance; Any performance measure possible	Parameter tuning

In terms of maintaining previous reference history, most of the algorithms maintain and use the recency of the last reference to the document, with the exception of the LNC-R-W3 algorithm that uses the k-th reference (Note the *Maintaining Reference History* column of Table 1). The LNC-R-W3 algorithm uses a strategy similar to the LRU-k algorithm that was proposed for buffer caching [7]. In contrast, the LUV algorithm, as we will show later, uses the reference recency history of all past references.

As for the reference frequency history, some algorithms simply use the frequency count, while others combine this information with the recency history that is maintained. The GD-SIZE algorithm, however, does not consider any frequency information. Again, note that the LUV algorithm uses the frequency count as well as the recency history of all prior references to a document.

Another aspect that may be considered in classifying replacement algorithms is the way in which reference history is maintained. Basically, there are two ways to maintain the reference history of documents. One is in-cache-history and the other is perfect-history. The in-cache-history method retains the reference history of only those objects that are in the cache. Hence, reference information is lost once the object is evicted. On the other hand, the perfect-history method retains the reference history of an object even after its eviction, allowing this information to be used when it returns to the cache. This method may offer considerable information about the reference behaviors, resulting in better prediction of future references than the in-cache-history method. However, it incurs more space and time overhead for maintaining the information of the evicted objects. To resolve this problem, perfect-history algorithms may choose to retain approximations of the reference history.

Breslau et al. show that perfect-history LFU outperforms in-cache-history LFU in terms of HR and BHR [13]. Recency history, i.e. reference time information, can also be maintained as either in-cache-history or perfect-history. In fact, most caching algorithms may be implemented using both in-cache-history and perfect-history methods. However, this paper basically deals with in-cache-history algorithms with the exception of some algorithms that inherently use perfect-history such as the LNC-R-W3 and LRV algorithms. Other algorithms shown in Table 1 are in-cache-history algorithms unless explicitly stated.

Other features of various Web caching algorithms, including advantages and weaknesses, are summarized in Table 1.

3. The LUV (Least Unified-Value) Algorithm

In this section, we describe the Least Unified-Value (LUV) replacement algorithm. LUV evaluates a Web document based on its retrieval cost normalized by the likelihood of it being re-referenced. This results in a normalized assessment of the contribution to the value of a document, leading to a fair replacement policy. Like most existing algorithms, LUV associates a value *Value*(i) to each document i in the cache and, when needed, replaces the document that has the smallest *Value*. *Value*(i) in LUV is defined as

$$Value(i) = Weight(i) \cdot H(i).$$

Weight(i) denotes the retrieval cost of a document per unit size and is defined as

$$Weight(i) = c_i / s_i$$

where c_i and s_i denotes the cost and the size of document i, respectively. c_i can be defined differently according to the performance measure of interest, and for the HR, BHR, and DSR measures, the c_i value used would normally be 1, the document size, and the download latency, respectively. For other measures such as social welfare as referred to by Kelly et al. [14], one can define c_i appropriately for the performance measure of interest.

$H(i)$ represents the likelihood of re-reference, that is, the worth of the document based on observations of past behavior. For the LUV algorithm, this is based on the recency and frequency history of all of the past references on document i. This notion is taken directly from the LRFU replacement policy [4]. Each reference to a document i in the past contributes to $H(i)$ and a reference's contribution is determined by a weighing function $F(x)$ where x is the time span from the reference in the past to the current time. For example, assume that document i was referenced at t_1, t_2, and t_3. Then, $H(i)$ at current time, t_c, is computed by

$$H(i) = F(\delta_1) + F(\delta_2) + F(\delta_3)$$

where $\delta_1 = t_c - t_1$, $\delta_2 = t_c - t_2$, and $\delta_3 = t_c - t_3$.

A formal definition of $H(i)$ is, then, given as

$$H(i) = \sum_{k=1}^{n} F(t_c - t_k)$$

where t_c is the current time, n is the number of references made to document i since it has been brought into the cache, and t_k is the time of the k-th reference. $F(x)$ would generally be a decreasing function as more weight should be given to more recent references. In this paper, we use the function that was chosen in the original LRFU presentation [4], that is,

$$F(x) = (1/2)^{\lambda x} \quad (0 \le \lambda \le 1).$$

```
if document i is already in the cache
{   Update the Value of i;
    Adjust the position of i in the heap appropriately;
}
else
{   Fetch document i from the original site;
    Give an initial Value to i;
    while(Size of i is larger than free space)
        Remove the root and reorganize the heap;
    Insert document i into the proper place in the heap;
}
```

Figure 1. Operation upon request for a document on the Web.

When λ is equal to 0, $H(i)$ simply counts the number of previous references and LUV is reduced to the sw-LFU [14], which is basically a weighted LFU. As λ increases, LUV gives more weight to more recent references. When λ is equal to 1, the original LRFU is reduced to the LRU algorithm, which considers only the last reference time. Then, at first glance, LUV may again be thought of as a weighted LRU similar to GD-SIZE [2]. However, this is not the case, though LUV indeed gives more weight to more recent references. Moreover, there is an important difference between LUV with λ=1 and GD-SIZE. Both of the algorithms increase the *Value* of a document when it is referenced, and decrease it as time progresses. This aging mechanism reflects the recency of past references in caching environments where non-uniform costs may be associated with the objects. While LUV applies the same decremental-*rates* to all of the documents in the cache, GD-SIZE applies the same decremental-*values*. For example, let the *Value* of documents A and B be 1000 and 10, respectively, at time t, and no reference is made to these documents up to time t+1. Then, when the same decremental-value of 1 is applied, as in GD-SIZE, the *Value* of A and B becomes 999 and 9, respectively, at time t+1. When the same decremental-rate of 0.1 is applied, as in LUV, the *Value* of A and B becomes 900 and 9, respectively. Though both methods are identical in philosophy, the same decremental-value method has a weakness that it may incur cache pollution when the cost of documents has large variations as it may be difficult to age documents with large costs in such environments.

4. Implementation of the LUV Algorithm

According to the description of the LUV algorithm given in Section 3, computing the *Value* of each document requires all of the past reference times. At first thought, this may seem impractical as space and time overhead for attaining and maintaining this information may seem infeasible. Moreover, *Value* of each document changes over time, and this necessitates recomputing the *Value* of all documents in the cache as time progresses. Here, we show that this is not the case, and that an efficient implementation is possible. Similar arguments were presented for the LRFU algorithm [4]. In this paper, we show that the original argument can be extended to the Web caching realm, where the *Weight* factor is incorporated into the document value calculations.

Property 1. Let the *Value* of document i at the n-th reference time t and the $(n+1)$-th reference time t' be $Value_t(i)$ and $Value_{t'}(i)$, respectively. Then $Value_{t'}(i)$ is derived from $Value_t(i)$ as follows:

$$Value_{t'}(i) = Value_t(i) \cdot F(\delta) + Weight(i)$$

where $\delta = t' - t$.

Proof. Let δ_k denote the time interval between the k-th reference time and the n-th reference time where $1 \le k \le n$. Then,

$$\begin{aligned}
&Value_{t'}(i) \\
&= Weight(i)\, H_{t'}(i) \\
&= Weight(i)\{F(\delta_1+\delta)+ \cdots +F(\delta_n+\delta) + F(0)\} \\
&= Weight(i)\{(1/2)^{\lambda(\delta_1+\delta)}+ \cdots +(1/2)^{\lambda(\delta_n+\delta)} + (1/2)^0\} \\
&= Weight(i)\{(1/2)^{\lambda\delta_1}+ \cdots +(1/2)^{\lambda\delta_n}\}(1/2)^{\lambda\delta} + Weight(i) \\
&= Weight(i)\{F(\delta_1)+ \cdots +F(\delta_n)\} F(\delta) + Weight(i) \\
&= Weight(i)\, H_t(i)\, F(\delta) + Weight(i) \\
&= Value_t(i)\, F(\delta) + Weight(i)
\end{aligned}$$

□

Property 1 states that the *Value* at the time of the $(n+1)$-th reference can be computed directly from the time of the n-th reference and the *Value* and *Weight* at that time. Property 1 implies that the space complexity of LUV is constant per document, thereby resolving the space complexity problem.

Table 2. Characteristics of traces used in the experiments.

Trace	Trace gathering period	Total Requests	Total Unique Requests	Total Mbytes	Total Unique Mbytes
DEC [11]	09/01/1996 – 09/22/1996	609951	306685	6415.64	3826.34
NLANR [12]	07/18/1999 – 07/31/1999	2688258	1405635	23051.80	14290.83

Lemma 1. Let the *Value* of document i at time t and t' ($t' > t$) be $Value_t(i)$ and $Value_{t'}(i)$, respectively. If there have been no references to document i between the time interval t and t', $Value_{t'}(i)$ is derived from $Value_t(i)$ as follows:

$$Value_{t'}(i) = Value_t(i) \cdot F(\delta)$$

where $\delta = t' - t$.

Proof. Suppose document i had been referenced n times before t, and let δ_k denote the time interval between the k-th reference time and t. Then,

$$\begin{aligned}
&Value_{t'}(i) \\
&= Weight(i)\, H_{t'}(i) \\
&= Weight(i)\{F(\delta_1+\delta)+ \cdots +F(\delta_n+\delta)\} \\
&= Weight(i)\{(1/2)^{\lambda(\delta_1+\delta)}+ \cdots +(1/2)^{\lambda(\delta_n+\delta)}\} \\
&= Weight(i)\{(1/2)^{\lambda\delta_1}+ \cdots +(1/2)^{\lambda\delta_n}\}(1/2)^{\lambda\delta} \\
&= Weight(i)\{F(\delta_1)+ \cdots +F(\delta_n)\}F(\delta) \\
&= Weight(i)\, H_t(i)\, F(\delta) \\
&= Value_t(i)\, F(\delta)
\end{aligned}$$

□

Lemma 1 states that the *Value* of document i can be computed directly from a recently computed *Value* and the time this *Value* was computed.

Property 2. If $Value_t(a) > Value_t(b)$ and neither a nor b have been referenced after t, then $Value_{t'}(a) > Value_{t'}(b)$ holds for any t' ($t' > t$).

Proof. Let $\delta = t' - t$. Then, from Lemma 1,

$$Value_{t'}(a) = Value_t(a)\, F(\delta) > Value_t(b)\, F(\delta) = Value_{t'}(b)$$

$$(\because F(\delta) > 0)$$ □

Property 2 shows that the relative ordering of the documents in the cache does not change if they are not re-referenced. This property, along with Property 1, allows LUV to be implemented by a heap structure, where the time complexity of operations on this data structure is $O(\log_2 n)$. Figure 1 shows the algorithm that is invoked upon a request for a document.

5. Experimental Results

In this section, we discuss the results from trace-driven simulations performed to assess the effectiveness of the LUV algorithm. We used two public traces: Digital Equipment Corporation Web proxy traces (DEC) and access logs of proxy caches at the National Lab for Applied Network Research (NLANR). DEC traces used in our simulations are local-client traces that are references made by clients within Digital's Palo Alto campus [11]. NLANR provides ten sanitized proxy cache access logs: *bo*1, *bo*2, *lj*, *pa*, *pb*, *rtp*, *sd*, *sj*, *sv*, and *uc*. Among these *sj* was used in our experiments [12]. Table 2 shows the characteristics of the traces. In the experiments, we filtered out some requests such as UDP requests, *cgi_bin* requests, and requests whose size is larger than the cache size used in the simulation.

The performance of LUV is compared with those of the LRU, LFU, SIZE, HYBRID, LRV, GD-SIZE, LNC-R-W3, MIX, and sw-LFU in terms of HR, BHR, and DSR. Before discussing the results themselves, we point out for clarity, some of the peculiarities of the presented results and algorithms.

- In case of the LRV algorithm, we analyzed each trace a priori and used the complete parameter value information to reflect the characteristics of the target traces. Note that this would not be possible in real life.
- For the GD-SIZE algorithm, it has been reported that using the *cost* (c_i) value of 1 instead of the document download latency results in a higher DSR [2]. It is not clear why this is so, but we think this is due to the decrements used in GD-SIZE as described previously. As GD-SIZE applies the same decrements to all of the documents in the cache, it is slow in aging documents that have high latencies possibly

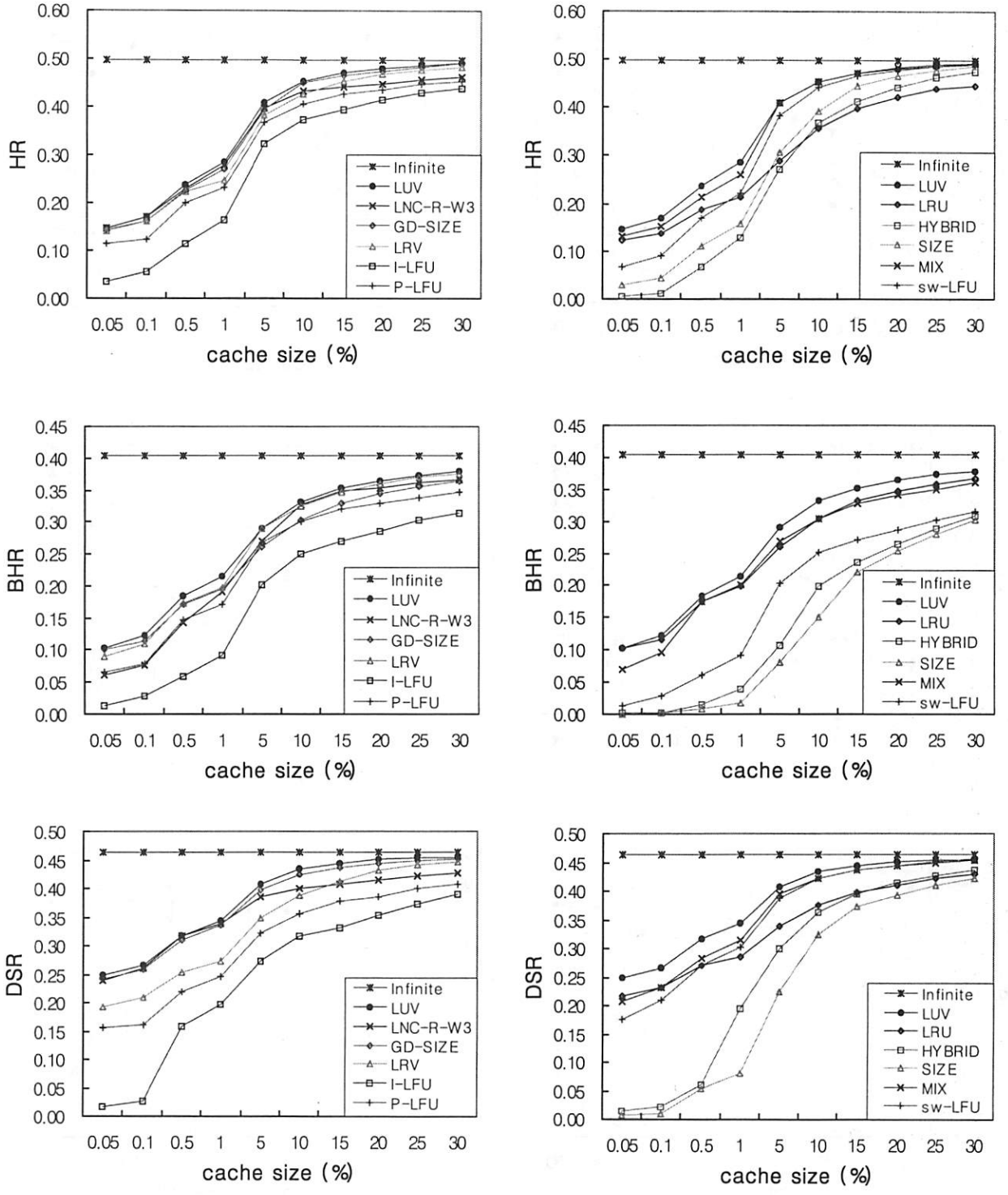


Figure 2. Comparison of LUV with other algorithms using DEC trace.

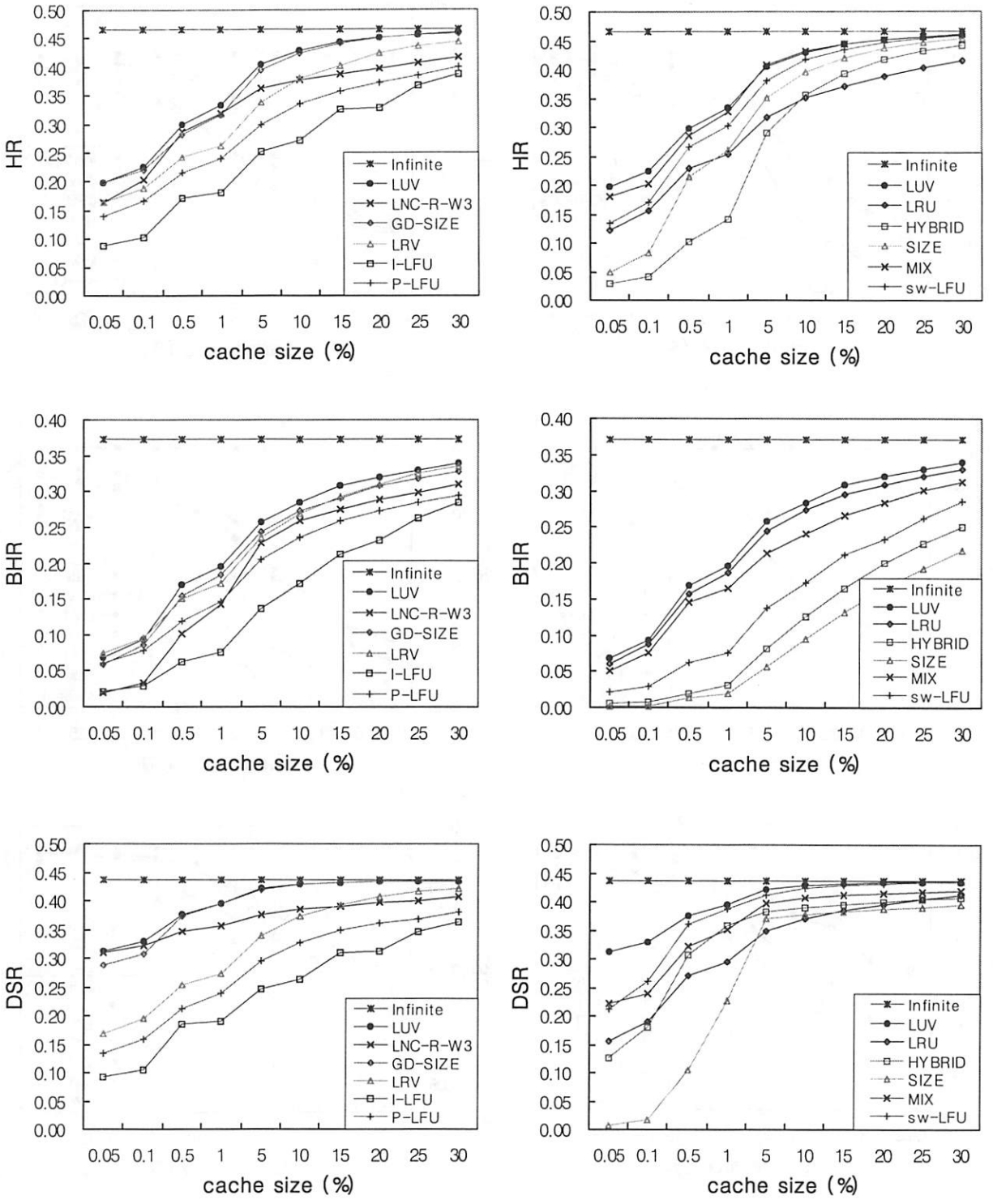


Figure 3. Comparison of LUV with other algorithms using NLANR trace.

leading to cache pollution. To deal with this situation, we experimented with both cost values, that is, 1 and the download latency, and selected the best results.

- LNC-R-W3 and LRV are inherently perfect-history algorithms [5,8]. For LFU, both in-cache-history and perfect-history methods are used. All other algorithms are in-cache-history algorithms.
- For GD-SIZE, sw-LFU, and LFU (both in-cache-history LFU and perfect-history LFU), the LRU order is used as a tie breaker. That is, when there is more than one document with the same *Value*, we select the victim by the LRU order.
- For the LUV algorithm, the λ value was tuned to reflect the characteristics of the given traces.

Figures 2 and 3 show the HR, BHR, and DSR of each algorithm as a function of the cache size for the DEC and NLANR traces, respectively. In the figures, *Infinite* represents the performance when the cache size is equal to *Total Unique Mbytes* in Table 2, which is essentially equivalent to having an infinite size cache. The x-axis, which is the cache size, is the size relative to the *Infinite* cache size. We simulated a wide range of cache sizes, from 0.05% to 30% of the *Infinite* cache size, to see the relation between the various cache sizes and the performance of each algorithm. Small cache size results, such as when the size is 0.05%, may be applicable to main memory caching, while results for large sizes may be appropriately used for disk caching. Note that the scale of the x-axis is not uniform. This arrangement is intended to show clearly the relative performances of the algorithms. Note that for clarity, we show the results in two groups of algorithms. The first (left-hand side) shows LUV, LNC-R-W3, GD-SIZE, LRV, and the two types of LFU, i.e. in-cache-history LFU (denoted by I-LFU) and perfect-history LFU (denoted by P-LFU), while the second (right-hand side) shows LUV, LRU, HYBRID, SIZE, MIX, and sw-LFU. From the results, the following observations can be made.

- For most cases, the LUV algorithm performs the best irrespective of the cache size for all performance measures, while the other algorithms give and take amongst each other at particular cache sizes and measures.
- Algorithms not considering the recency history perform worse than recency-based algorithms when the cache size is small. The performance of the SIZE, HYBRID, and LFU-based policies is consistently inferior and the gap is quite wide for most of the cache sizes that were studied. The only range that they come close is when the cache is large. Note that these algorithms do not consider the recency of past references at all, which is primarily considered in other algorithms compared in our experiments.
- We find that generally, frequency-based algorithms are inferior to recency-based replacement algorithms. However, as implied by the superior performance of the LUV algorithm, a good combination of recency and frequency can lead to even better performance.
- MIX shows good performance for all performance measures, though the performance gap between MIX and LUV is somewhat wider for smaller cache sizes. Recall, however, that the MIX algorithm requires $O(n)$ time complexity.
- Though perfect-history LFU shows better performance than in-cache-history LFU for all cases, there is no significant advantage of other perfect-history algorithms, that is, LNC-R-W3 and LRV over in-cache-history algorithms. In fact, even with perfect-history, none of these algorithms performed better than the LUV algorithm. Moreover, LNC-R-W3 requires $O(n)$ time complexity and LRV requires trace analysis overhead.

6. Conclusion

In this paper, we presented a Web cache replacement algorithm called LUV. LUV evaluates a Web document based on its retrieval cost normalized by the likelihood of it being re-referenced. This results in a normalized assessment of the contribution to the value of a document, leading to a fair replacement policy. Unlike most previous algorithms, LUV estimates the re-reference likelihood of a document by its full reference history during cache residency. Despite this, we showed that LUV allows for an efficient implementation in both space and time complexities. The space needed to maintain the reference history of a document is only a few bytes and furthermore, the time complexity of the algorithm is $O(\log_2 n)$, where n is the number of documents in the cache. Through trace-driven simulations we showed that the LUV algorithm performs better

than existing algorithms for various performance measures for a wide range of cache configurations.

One direction for future research is to model the distribution of Web requests seen by Web proxy caches considering both recency and frequency. Breslau et al. show that an independent request stream following a Zipf-like distribution is sufficient to model the requests seen at Web proxies [13]. However, our experimental results show that temporal locality is also important. Hence, we are attempting to model Web requests considering both the recency and frequency information to simultaneously reflect the reference popularity and the temporal locality. This will allow us to fix a priori the control parameter λ, which currently has to be determined through off-line experiments, according to the corresponding model.

Also, in this paper, we only considered the in-cache-history LUV algorithm, which requires low overhead in both time and space. However, because only a few bytes of information is required for each object, perfect-history LUV may improve performance even more without incurring much system overhead. We are currently conducting experiments to validate this conjecture.

Acknowledgment

We would like to thank NLANR and DEC for making their proxy traces available. Without their generosity this paper would not exist. Many thanks to the anonymous reviewers for their very helpful comments.

References

[1] M. Abrams, C. Standridge, G. Abdulla, S. Williams, and E. Fox, "Caching proxies: Limitations and potentials," *In Proceedings of the 4th International WWW Conference*, pp. 119-133, 1995.

[2] P. Cao and S. Irani, "Cost-Aware WWW Proxy Caching Algorithms," *In Proceedings of the USENIX Symposium on Internet Technology and Systems*, pp. 193-206, 1997.

[3] S. Glassman, "A caching relay for the World Wide Web," *Computer Networks and ISDN system*, vol. 27, no. 2, pp. 165-173, 1994.

[4] D. Lee, J. Choi, J. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. Kim, "On the Existence of a Spectrum of Policies that Subsumes the LRU and LFU Policies," *In Proceedings of the 1999 ACM SIGMETRICS Conference*, pp.134-143, 1999.

[5] P. Lorenzetti, L. Rizzo, and L. Vicisano, "Replacement Policies for a Proxy Cache," http://www.iet.unipi.it/~luigi/caching.ps.gzeunis, 1996.

[6] N. Niclausse, Z. Liu, and P. Nain, "A new and efficient caching policy for the world wide web," *In Workshop on Internet Server Performance (WISP'98)*, 1998.

[7] E. O'Neil, P. O'Neil, and G. Weikum, "The LRU-*k* Page Replacement Algorithm for Database Disk Buffering," *In Proceedings of the 1993 ACM SIGMOD Conference*, pp. 297-306, 1993.

[8] P. Scheuermann, J. Shim, and R. Vingralek, "A Case for Delay-Conscious Caching of Web Documents," *In Proceedings of the Sixth International WWW Conference*, 1997.

[9] S. Williams, M. Abrams, C. Standridge, G. Abdulla, and E. Fox, "Removal policies in network caches for world-wide web documents," *In Proceedings of the ACM SIGCOMM '96*, pp. 293-305, 1996.

[10] R. Wooster and M. Abrams, "Proxy caching that estimates page load delays," *In Proceedings of the Sixth International WWW Conference*, 1997.

[11]ftp://ftp.digital.com/pub/DEC/traces/proxy/locallist v1.2.html

[12] ftp://ircache.nlanr.net/Traces/sj_sanitized-access_990718.gz ~ sj_sanitized-access_990731.gz

[13] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker, "Web Caching and Zipf-like Distributions: Evidence and Implications," *In Proceedings of Infocom '99*, 1999.

[14] T. Kelly, Y. Chan, S. Jamin, and J. MacKie-Mason, "Biased Replacement Policies for Web Caches: Differential Quality-of-Service and Aggregate User Value," *In Proceedings of the Fourth International Web Caching Workshop*, 1999.

[15] L. Rizzo and L. Vicisano, "Replacement Policies for a Proxy Cache," *Technical Report UCL-CS RN/98/13*, 1998 (available at http://www.iet.unipi.it/~luigi).

Providing Dynamic and Customizable Caching Policies*

J. Fritz Barnes Raju Pandey
Parallel and Distributed Computing Laboratory
Computer Science Department
University of California, Davis, CA 95616
{barnes, pandey}@cs.ucdavis.edu

Abstract

Web caching has emerged as one solution for improving client latency on the web. Cache effectiveness depends on the policies used to route requests to other caches and servers, to maintain up-to-date web objects and to remove objects from the cache. Traditional caches apply one set of policies, which determines the efficiency as well as the effectiveness of the caches. This set of policies often does not exploit the diversity inherent in different web objects, caches and clients. Policies that do exploit this diversity result in convoluted caching policies that attempt to combine multiple policies and guess at the unknown characteristics of web objects, caches and clients.

In this paper, we present an extensible caching infrastructure in which cache administrators, servers, and end users can customize how web objects are cached, replaced, and kept consistent. The infrastructure includes a domain-specific language, CacheL, for defining customizable caching policies that can be changed dynamically. Analysis of our prototype, PoliSquid, shows the benefits of the infrastructure for variable coherency policies, localized removal policies, and early removal of objects from servers.

1 Introduction

Web caching [15, 21, 8] improves client latency through the use of intermediate servers or caches, that accept HTTP [10] requests from clients. When a request is made to a cache, the cache returns a copy of the web object being requested. If a replica does not exist, the cache contacts other caches or the origin server to retrieve a copy. Caches improve web performance by migrating web objects closer to clients. Further, by not contacting the server, caches reduce origin server load.

Caches use several policies for cache management. Cache policies determine:

- which objects and when to pre-fetch (*pre-fetch policy*),
- how requests for objects are routed (*routing policy*),
- where objects are placed in a cache hierarchy (*placement policy*),
- how cache objects are kept fresh (*coherency policy*), and
- which objects to remove when the cache is full (*removal policy*).

Research on caches has focused primarily on developing optimum policies for all users and objects. For instance, several removal policies such as Least Recently Used (LRU), Least Frequently Used (LFU), Perfect LFU [3], and removal of large objects from caches (SIZE) [23] try to reduce bytes requested from the server and client latencies. These policies depend on a single attribute of the web object. For instance, SIZE removes large objects from the cache which make room for multiple smaller objects. As a result, caches hit rate improves. However, there are a variety of cache performance measurements. Often a single web object attribute will not necessarily improve all measurements. The SIZE algorithm improves the hit rate, but may degrade client latencies

*The authors are partially supported by the Defense Advanced Research Project Agency (DARPA) and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-97-1-0221.

due to the high network overhead of uncached large documents.

Several object replacement algorithms have been developed that extend the single attribute algorithms by weighting multiple attributes or applying cost functions. These algorithms include: a HYBRID policy [24] that uses a weighted combination of object attributes; Greedy-Dual-Size [4] that uses an appropriate cost function; and biased replacement policies [12] that use a combination of weights based on the origin server and LRU. These policies provide more flexibility in optimizing multiple cache performance measurements. However, knowledge about the cache environment, such as the bandwidth of the nearby network topology, cannot be explicitly represented in these algorithms. Parameterizations and cost functions are not sufficiently general enough to define policies for specific environments.

In addition to dependence on multiple parameters, cache policies may also depend on the semantics of the cache objects. For instance, if we consider objects in a cache, we observe that there is a large variance in hit rates for objects with different content types. Image requests generate a hit rate that is about 4-5 times greater than the hit rate for hypertext files [18]. In other words, a single set of policies cannot be applied to a diverse population of objects, caches and clients.

Further, cache policies are often dynamic in nature. An example is the applicability of cooperative algorithms in different operating conditions. Cooperative caching improves performance by sharing objects across multiple caches. This effectively increases the cache store size. Additionally it increases the number of potential clients, which increases the probability that two clients will request the same object. An analysis of cooperative algorithms [14] concludes that the overhead in communication between cooperating caches may outweigh the benefit of additional hits provided by a neighboring cache. Dynamic policies provide better support in these circumstances. Cooperation should be used when the overhead is inconsequential compared to the reduced bandwidth and increased hit rates. However, cooperation should be turned off when the overhead becomes too great. Therefore, caching policies should be dynamic in response to the ever-changing pattern of web requests.

Thus, what is needed is a caching infrastructure that allows both clients, servers, and caches to specify and customize policies to suit semantics of objects, variations in parameters, and the dynamic behavior of the web. In this paper, we present such a caching infrastructure. In this infrastructure, a cache policy is defined by registering a set of actions with a set of events. The events denote entry points for applying caching policies, whereas the actions implement specific caching policies. The infrastructure includes a domain specific language, called CacheL, for specifying the actions. The infrastructure also allows caching policies to be changed dynamically in order to take advantage of the changing environment.

We have implemented the infrastructure as a cache simulator, DavisSim and as part of a web cache, PoliSquid. We evaluate the benefit of: allowing variation in client tolerance for fresh documents; customizing removal policies to the network topology and using object semantics to remove objects earlier than the standard removal policy. The results of the experiments demonstrate that the customizations help improve cache performance. Finally, we investigate the overhead associated with adding extensibility to a caching system, Squid [20]. The results show that the overhead is moderate (about 8.5%) and can be improved significantly with aggressive optimizations.

This paper is organized as follows: We present a taxonomy of the different caching policies and how they can be customized in Section 2. In Section 3, we describe an infrastructure for providing these customizations. In Section 4, we briefly present the design of a caching simulator and our prototype implementation. We present the performance analysis of our infrastructure in Section 5. We describe related work in Section 6. We conclude with a summary and discuss future work.

2 Customizable Caching Policies

In this section, we look at the various cache policies and how they can be extended. The behavior of a cache is defined by a set of cache policies: prefetch, routing, placement, coherency, and removal policies. In Table 1, we list the different policies and the applicable customizations. Below we focus on the pre-fetch, routing, and coherency policies in more detail.

Policies	Creator of customized policy		
	Client	Cache	Server/Web Object
pre-fetch policy	client-side pre-fetch	traditional pre-fetch	push-caching
routing policy	routing based on clients	complicated peering arrangements	mirroring servers
placement policy	not-applicable	cooperative caching	push-caching
coherency policy	personalize user tradeoffs	policy for shared documents	specify when objects expire
removal policy	not-applicable	take advantage of costs realized at the cache	take advantage of semantic content
miscellaneous	content transducers	measurement & tuning	protocol extensions

Table 1: Analysis of different caching policies

Pre-fetching policies request objects before they are requested by a client. Pre-fetching increases the hit rate because the first document access might result in a hit. However, pre-fetching can increase the amount of bandwidth requested by the server if a pre-fetched object is never accessed. The effectiveness of pre-fetching depends on how well the pre-fetching policy can predict which objects will be accessed and therefore should be retrieved in advance.

Pre-fetching can be used in different ways. A client might use pre-fetching to load objects from the cache into the browser. This may be of particular interest in reducing the latency due to slow dialup connections. Allowing pre-fetch customization allows the cache to support different browsers using different schemes to perform pre-fetching of documents into the browsers.

Caches could be customized to accept objects pushed out from servers. This provides a technique whereby servers can perform push-caching of popular documents. As a result, fewer requests will be served at the origin server reducing server contention and improving performance.

Routing policies determine how a cache retrieves an object. Clients, caches and servers might each use the routing policy in a different manner. The manner in which the cache routes outgoing requests might be determined by the client. Let us consider a network that supports differentiated services. Clients could specify routing policies that use different priorities of services. A cache administrator, on the other hand, might specify a routing policy that allows cooperation among multiple caches. A routing policy provided by a web site might be used to alternate requests between different mirrors of that web site. Even better, the routing policy might determine the optimal mirror the cache should contact.

Coherency policies determine how a cache responds to a request for an object in the cache. The coherency policy decides either to consider the object fresh or stale. In the case of stale objects, the cache consults with the server to verify that the copy is up-to-date. A commonly used algorithm in deciding coherency is the TTL algorithm [6]. This algorithm determines whether an object is fresh by evaluating the equation:

$$T_{now} - T_{in} < k\,(T_{in} - T_{last\_mod}) \qquad (1)$$

where T_{now} is the time of the request, T_{in} is the time when the object entered the cache or was last verified, and T_{last_mod} is the time when the object was last changed. If the boolean expression is true than the document is considered fresh. Different clients may desire different values of k. Larger values of k result in fewer validation checks which would increase hit rates with a tradeoff of sometimes returning stale objects. Caches set the global default coherency policy. Additionally, a cache might place limits on the range of client customization. Servers can specify an invalidation scheme to use, or an algorithm for deciding coherency that takes into account their object/server specific characteristics. For example, a server specified coherency policy might take advantage of the fact that all of its objects are only updated in the morning.

In the next section, we describe in more detail our infrastructure for specifying these customized policies.

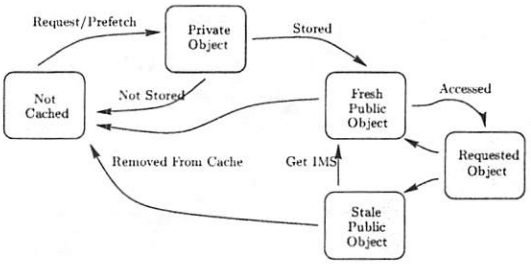


Figure 1: States of a document in a cache

3 Customizable Infrastructure

In this section, we discuss our caching infrastructure for creating customizable policies. Our goal is to create a caching system in which clients, cache administrators, or web object authors can customize caching policies. The caching infrastructure provides the ability to specify different routing, placement, coherency, and removal policies for objects. We first discuss an event-based model, which provides the technique for specification of policies. These policies are written in a domain-specific language, CacheL. We next describe the architecture used to attach policies to web objects. We conclude this section with an example that uses CacheL to provide a customized removal policy.

3.1 Event-based Architecture

The caching infrastructure is based on subdividing a policy into actions that are taken when specific events occur. In Figure 1, we show the pertinent states in the lifetime of a web object. The arcs in the figure represent events that cause an object state to change.

For example, we will discuss the changes in state of the web object, `http://wonderland.net/tea-party.html`, as first the Dormouse and later Alice request the tea party object. Initially, the tea party object exists in the Not Cached state. When the Dormouse makes a request for the tea party object, the routing policy is used to determine how the cache retrieves a copy of the tea party. This object enters the Private Object state. Now, the placement policy determines whether we store the tea party object in the cache. If we store the object it enters the Fresh Public Object state. Later, when Alice requests the tea party object, we enter the Requested state. At this point the coherency policy determines whether the document is fresh or stale and, if stale, checks whether the original has changed using a Get If-Modified-Since (IMS) request.

The amount of space available in the cache also affects the state of web objects. When the cached bytes exceed capacity, the cache uses a removal policy to determine the best objects to be removed. These objects leave the Fresh or Stale Public Object states and return to the Not Cached State.

3.2 CacheL

CacheL defines policies with a policy script. A policy script specifies a set of actions to take when different cache events occur. An action consists of a set of cache operations and expressions. Cache operations include contacting other caches and servers, changing the state of local objects, and setting alarms to schedule an action in the future.

The language supports simple variables; relational, arithmetic and logical expressions; conditional execution; set, array, numeric, and string data types; and iteration over sets. It also supports facilities for manipulating date/time values, as well as MIME headers associated with objects.

3.2.1 Cache Events

The infrastructure provides a predefined set of events. We describe what causes these events as well as required and optional cache operations that should be associated with these events.

Route events occur when objects must be retrieved. The action associated with a Route event should send an HTTP request to a cache or the origin server, return an alternate document in the cache, or respond with an error.

New-Store events occur after new documents are retrieved or Get IMS requests for stored documents return new documents. Associated actions should decide whether the object should become a Fresh Public Object or return to the Not Cached state.

Access-Inline events occur before responding to a request made for an object in the cache. Associated actions should handle coherency policies.

Actions can also modify the MIME headers of the object requested.

Access-Offline events are triggered by a request but handled independent of the response. Actions that do not affect the response should be attached to this event. For instance, a counter could be used to keep track of object accesses.

High-Water events occur when the bytes stored in the cache exceed a predetermined level. Associated actions should purge one or more objects from the cache.

Purge events occur when a document is removed from the cache. This event allows actions to inform another party when an object is removed, or to adjust policy-specific state.

Timer events occur as a result of policy-defined alarms. Actions are policy dependent.

3.2.2 Cache Operations

We enumerate some of the pertinent cache operations:

CacheFetch(URL) requests a web object from the object's originating server.

CacheFetchIMS(URL) performs an If-Modified-Since request for the given web object.

CachePost(URL, Data) contacts a host with a post request and includes the Data in the request. This is used primarily for server-specific policies to pass information back to their originating server.

CacheICPFetch(HOST, URL) requests a web object from another cache using the Internet Caching Protocol.

CacheResponse(URL) sends a local copy of URL to the client as a response to the client's request.

CachePurge(URL) removes the specified object from the local store.

CacheStore(URL) instructs the cache to store the object that is being requested.

CacheSetTimer(Date, Time) sets an alarm.

CacheLog(Message) provides a debugging mechanism that writes the given message to the cache's log file.

3.3 Cache Policy Management

The infrastructure provides support for cache clients, cache administrators, and the web site author to specify caching policies. Cache administrators specify policies locally and can refuse to allow servers to modify policies. Currently, we require a cache administrator to specify client policies by mapping client identifiers to policies. However, in future work we would like to investigate alternative techniques for clients to post policies.

In order to specify policies, cache administrators construct a configuration file that contains a list of web objects and the URLs of policy scripts. The Web objects can occur multiple times within the configuration file for different policies. Cache administrators can utilize wildcards for easy specification of policies that apply to multiple objects and hosts.

In addition to allowing cache administrators to define policy mechanisms, server administrators can define policies relevant to a web object on their server. Policies are specified by adding an X-Cache-Policy MIME header that specifies the URLs of the policy scripts that apply to the object. When a cache retrieves the object with an X-Cache-Policy header, it retrieves the policies if they are not already in the cache. Then it applies the actions for the web object so that the desired policies can be enforced.

3.4 Removal Policy Example

We provide an example situation where a cache administrator wishes to customize the cache removal policy. This example provides a concrete demonstration of actions written in CacheL. More examples can be found on our web page `http://pdclab.cs.ucdavis.edu/qosweb/CacheL/` and in a previous paper [1].

In this example, the caching administrator controls caches distributed across several different work sites. A cache exists at each of these sites and a single connection to the Internet is provided. The bandwidth to the Internet is limited, and the bandwidth between the cache locations is plentiful. This scenario is shown in Figure 2, where Sprockets International has offices in San Jose, Boulder, and Boston, with

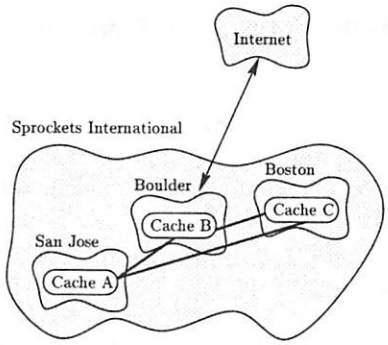


Figure 2: Cooperative Caches located at different sites for Sprockets International

```
Event: Initialize
   CreateList( "Unique" );
   CreateList( "Duplicate" );

Event: New-Store( obj )
   if ( CacheDirectoryLookup( obj ) == "NotFound" )
       ListAdd( "Unique", obj );
   else
       ListAdd( "Duplicate", obj );
   endif

Event: High-Water
   if ( ! IsListEmpty( "Duplicate" ) ) {
      obj = GetLast( "Duplicate" );
      if ( CacheDirectoryLookup( obj ) != "NotFound" )
         CachePurge( obj );
         ListRemove( "Duplicate", obj );
      else
         ListRemove( "Duplicate", obj );
         ListAdd( "Unique", obj );
      endif
   else
      obj = GetLast( "Unique" );
      CachePurge( obj );
      ListRemove( "Unique", obj );
   endif
```

Figure 3: Actions used to implement a specialized removal policy

a single connection to the Internet. As a result of the limited bandwidth to the Internet, the cache administrator would like to favor removal policies that evict objects that are cached in multiple locations, thus maximizing the total number of objects cached.

We show an implementation of this policy in Figure 3. The policy employs several actions associated with different events. Our infrastructure provides multiple linked lists and priority queues that can be used for implementing the removal policy. To implement the desired cache policy, we assume that each cache has a directory containing objects cached by nearby servers [19, 17, 9]. The CacheDirectoryLookup operation consults the local entries in the cache table and returns the neighbor that caches an object or NotFound.

The implementation associates actions with the New-Store and Highwater events as well as initially creating the linked lists that will be used by this policy. When an object is first stored in the cache, the removal policy determines whether another cache already caches the object. If the object is already cached it is stored in the Duplicate linked list; otherwise it is stored in the Unique linked list. When a Highwater event occurs, the cache attempts to purge objects that are cached by other caches. The Highwater action requires that we doublecheck that the objects are still cached by a neighboring cache since one of the neighbors may have removed it.

4 Infrastructure Design and Implementation

We have implemented two caching systems that are based on our extensible caching infrastructure. The first, DavisSim, is an event-based cache simulator. The second, PoliSquid extends the popular Squid [20] web cache by adding the ability to specify cache policies. We have used an event based design for implementing the caching systems. The event-based design matches well with the implementation of Squid, which uses events to handle asynchronous I/O. When an event occurs the caching infrastructure will determine whether an action is attached to that event and execute all attached actions. Below we briefly describe the CacheL interpreter, DavisSim, and PoliSquid.

CacheL Interpreter: We have implemented an interpreter that is invoked for executing actions specified in CacheL. Our current implementation uses a recursive descent interpreter and does not transform the input into an optimized abstract representation. As a result, each execution of an action requires a parsing step. In future versions, we plan to store the abstract representation within the cache in order to avoid the overhead of parsing a policy script every time it is executed.

DavisSim: is based on the Wisconsin Cache Simulator[1]. It uses a pre-processed web trace. This web trace contains the time of request and server, document and client identifiers. Additional information

[1] Available at: `http://www.cs.wisc.edu/~cao/webcache-simulator.html`

about the object, such as last modified time, size, and perceived latency, are included in the trace. Each request in the trace causes either a New-Store event if an object is not already cached, or an Access-Inline event if the object is cached. We do not support Routing events at this time because DavisSim does not simulate cooperative caching.

The cache operations allow scripts to store objects in the cache and purge objects to make more space. Internally the simulator maintains statistics about true/false hits, true/false misses, and latencies. A true hit occurs when the object being requested hits in the cache and the object has not been modified from the version stored in the cache. If the object was modified it is considered a false hit. Similarly, if an IMS request is generated and the object has been modified we refer to this as a true miss, otherwise a false miss. In some of our experiments, we utilize CacheL to provide additional statistics. This is useful when we want to keep statistics about two different types of objects.

PoliSquid: allows us to load CacheL scripts specified by the cache administrator in a configuration file. We store policy files as another object in the cache. When an event occurs for an object, the policy executor looks up the URL associated with that script and then passes this to the interpreter.

5 Analysis

We now analyze the performance behavior of the extensible caching infrastructure. The primary goals of analysis are:

- *Do caches benefit from supporting customizable policies?*
- *What is the overhead of adding extensibility to a caching system?*

In the remainder of this section, we describe the experiments we have conducted in order to address the above questions.

5.1 Effectiveness of Customizable Caching Policies

In this subsection, we analyze the performance of the extensible caching infrastructure when the clients, caches and origin servers implement different caching policies. We consider the benefit of client, cache and server customized policies through three experiments. In the first two experiments, we use DavisSim to evaluate the effects on the cache. We examine a weeks worth of the DEC Squid traces[2] in our simulations. We further refine the dataset for coherency experiments by removing files without last modified times. These requests are ignored because we want to get an idea of how changing factors in the TTL computation affect coherency. We also remove those documents that are requested only once. Experiments involving removal policies use all requests in the week, and use an optimal prediction of whether a file has changed.

We make the following assumptions in the cache simulator. We use the LFU algorithm by default unless otherwise stated for cache removal. We use a value of $k = 0.5$ in the TTL algorithm (Equation 1) to decide coherency unless otherwise stated. If a new request exceeds the size of the cache, then the request is considered a miss and is not stored in the cache. The simulator creates high water events when the size of the documents stored in the cache exceeds 95% of the cache size. The cache discards documents until the stored documents occupy less than 90% of the cache size.

5.1.1 Client-Customized Policies

We perform two experiments that look at the effects of different clients selecting a different factor in the TTL calculation of document freshness. The first experiment explores the benefit of allowing clients to specify different factors in the TTL calculation. In this experiment, we subdivide the clients into two groups: strict clients and lax clients. The strict clients have minimal tolerance for web objects that have changed. As a result they utilize a small constant ($k = 0.001$) in the TTL calculation. However, this small constant penalizes the lax clients that can tolerate some old information. In Figure 4, we show the results of the experiment, varying the

[2]Provided by Compaq at: `ftp://ftp.digital.com/pub/DEC/traces/proxy/webtraces.html`

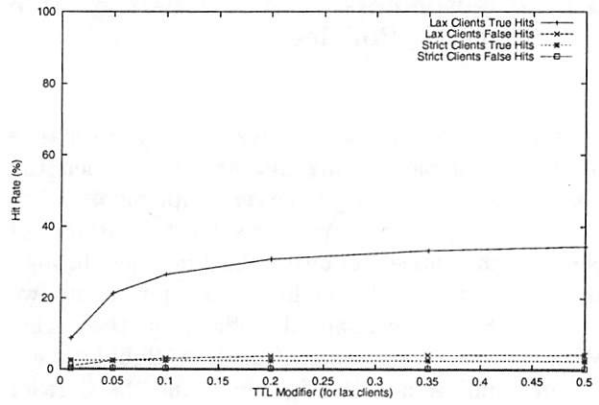


Figure 4: Cache Hit rates when different coefficients are used for lax clients.

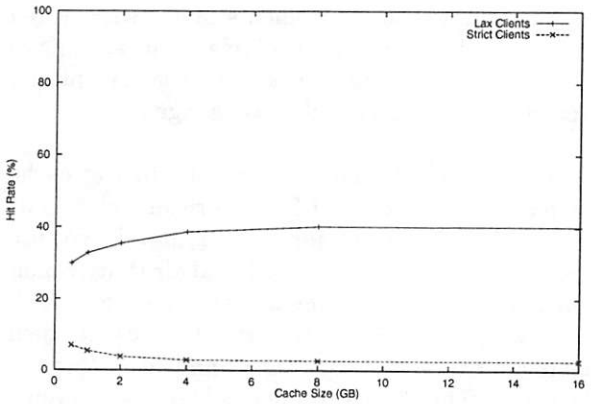


Figure 5: Change in latency when half of the cache clients utilize a restrictive caching policy and the others use a variable caching policy.

TTL constant for the lax clients. The graph plots the true/false hit rates achieved by the clients. The strict clients have a low true hit rate because they require many IMS requests to verify coherency, or false misses. As a tradeoff however, they minimize the number of false hits to about 0.3%. In contrast, the lax clients achieve hit rates 10 times the level of the strict clients at the cost of a higher false hit rate of 4.5%. This experiment demonstrates that it is desirable to allow clients to specify their desired level of coherency.

In the second experiment, we look at the effects of changing the cache size on multiple coherency policies. In this experiment we use a constant k of 0.5 for the lax clients. The results of the experiment, shown in Figure 5, demonstrate that hit rates improve as the cache size increases for the lax clients. However, we see the hit rates decrease for the strict

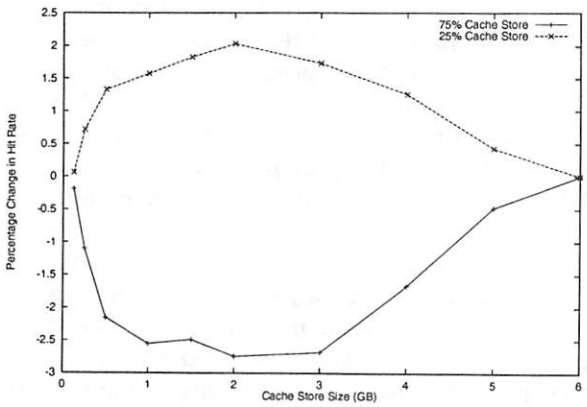


Figure 6: Percentage Differences in hit rates for objects subdivided by incoming network

clients. The reduction in the hit rates for the strict clients is due to the greater reuse of the objects by the lax clients. When the cache is small, it is more likely that an object will not exist in the cache and therefore the the lax clients will miss more often, resulting in objects that are fresh for small k values. However, as the cache size increases, the lax clients requests will generate more hits, thereby decreasing the number of objects that are fresh for small k values.

5.1.2 Cache-Customized Policies

In this experiment, we explore cache-customized policies that take advantage of the network environment near a cache. We assume that the cache's host has a high-bandwidth and low-bandwidth network link. The cache administrator would like to allocate a greater percentage of the cache to documents arriving over the low-bandwidth link in order to increase the hit rate on that link.

DavisSim uses a user-defined function that determines whether incoming requests traverse the high-bandwidth or the low-bandwidth link. We subdivide the cache into two portions: we use 75% of the cache store for the slow link, the rest for the other link. The resulting hit rates are shown in Figure 6. The graph represents the difference in the hit rate when the cache is subdivided to hit rates when the cache is not subdivided and demonstrates that at small and large cache sizes there is not much benefit to using subdivided caching policies. However, at medium cache sizes the hit rate for the slow link is improved by 2%, with a decrease in the hit rate for

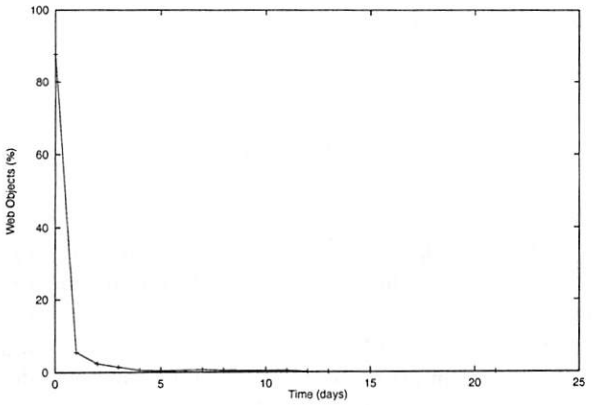


Figure 7: Percentage of objects accessed within x days of the original access

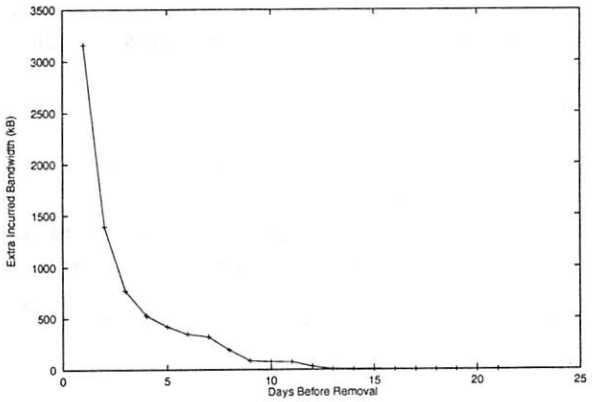


Figure 8: Extra Bandwidth necessary if files are removed early

documents over the fast link. These results, demonstrate that subdividing the cache can adjust the hit rates for different sets of objects when the cache is large enough that objects are not removed immediately and not so large that all objects fit within the cache.

5.1.3 Server-Customized Policies

In this experiment we explore the benefits of using information based on the server's web content for customizing caching policies. The idea in this experiment is to consider a web newspaper that adds new articles each day. We use NLANR logs from December 21, 1998-February 2, 1999 to investigate the unique behavior of news articles and images from the ABCNews web site[3]. The files in our trace represent 29 MB of requested objects. In Figure 7, we plot the time over which a document continues to be requested. We notice that 87% of the web objects are not accessed after 24 hours from the initial request. Therefore, by taking advantage of the object semantics to remove objects early, we can make room for other objects in the cache. To evaluate the benefits, we apply a specialized removal policy that purges documents from the cache after a given number of days have passed. Figure 8 shows the additional bandwidth required to access files that were removed early. If documents are purged after 1-2 days, there will be .5-1.5 MB of documents that will be requested that were previously cached. Most of this bandwidth comes from a few files. A smarter policy would allow exceptions to the one-day purge rule for these few files. We conclude that using the document semantics can provide a benefit in decreased cache utilization by files that won't be accessed again and a slight decrease in hit rate for a few popular pages.

[3] `http://www.abcnews.com/`

5.2 Overhead of Customizable Policies

We now describe an experiment for assessing the overhead of using extensible policies. We wish to measure the overhead of generating events, interpreting CacheL scripts, and taking appropriate actions. Our experiment involves specifying a script to handle Access-Inline events. These events occur every time a hit occurs for an object. Therefore, when we ran the experiment we specified a trace where we achieve a 100% hit rate after a warmup period. In our experiment we wrote the coherency policy contained within Squid, including special cases, as a script, which consisted of 20 lines of CacheL code.

We use three 350 MHz Pentium II PCs running the Linux operating system to perform the experiments. Two machines were used as client/server processes for the Polygraph[4] web proxy performance benchmark. We performed an experiment to measure the latency observed in accessing a 4KB file. We use a single process accessing the files with a short delay between the requests. The experiment measures the mean response latency.

We find that the latency of requests using PoliSquid to calculate the coherency policy incurs an 8.5% overhead compared to the latency for Squid. Fig-

[4] Polygraph is available at: `http://www.ircache.net/Polygraph/`

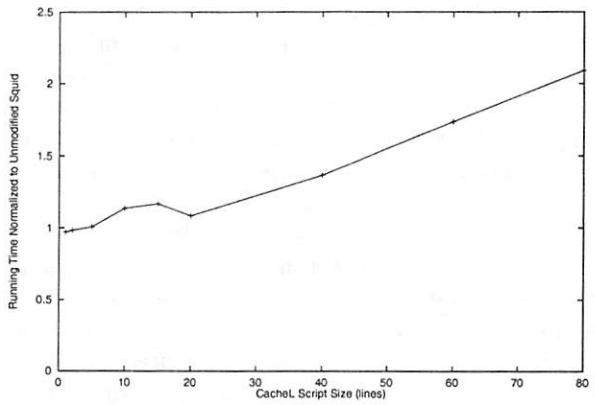


Figure 9: Response overhead of PoliSquid normalized to the running time for Squid using the same requests.

ure 9 contains the results as we vary the number of lines in the coherency policy script. The non-linear result is due to the manner in which different length scripts were created. Scripts were created as follows:

- **1-20 lines**: repetitive TTL calculations (Equation 1), the most expensive statement in the coherency policy script;
- **20 lines**: duplicating squids hard-coded policy in squid;
- **20+ lines**: inserting additional TTL calculations.

We note that an actual coherency policy would be shorter than 20 lines because it would be customized to the attributes of the object. For example, it would not use conditional statements to determine which coherency policy to employ, rather different policies would be attached depending on the object's attributes. In actual operation, the overheads would be less apparent due to larger file sizes, and I/O overheads due to network and disk accesses by a large group of users. In addition, our modified version of Squid has not been optimized, which we believe can reduce the overhead substantially.

6 Related Work

There are two relevant areas of previous work that relate to extensible web caching: techniques to allow customization of caches and schemes that give greater control over caches to servers.

6.1 Cache Customization

The Squid web cache [8, 22, 20] is one of the more popular web caches currently deployed in caching architectures. Squid supports composition of and parameterization of policies. For example, Squid uses expiration, time-to-live (TTL), or constant lifetime coherency policies. The policy depends on whether an expiration or last-modified time was included with the object. Squid allows the cache administrator to customize caching policies through modification of parameters and weights. This differs with our infrastructure, which allows a cache to specify different policies using an interpreted language.

An alternative for communicating between caches and organizing the hierarchies is discussed by Zhang et al. [25]. Their adaptive technique for organizing caches uses separate hierarchies for different servers, to avoid the overload that would occur at a single server. The adaptive cache configures itself and allows hosts to enter and exit cooperation.

Nottingham's work in optimizing object freshness controls [16] considers techniques to avoid object validation. He evaluates optimum parameters for freshness calculation that depend on the object type and the location from which the object was retrieved.

Other languages have been proposed for use on the web. One of these is WebL [13], used for document processing. The goal of this system is to provide both reliable access to web services through the use of service combinators and techniques for gathering information and modifying documents through markup algebra. WebL is intended for a different purpose than CacheL, and in fact combining the two may be fruitful: it would allow caches to make simple customizations of web pages instead of forcing these requests to be handed by the origin server.

6.2 Server Control

Caughey et al. [7] describe an *open caching* architecture for caching web objects that exposes the

caching decisions to users of the cache. Open caching allows both clients and servers to customize the caching infrastructure. Customization is performed through object-orientation of resources.

Push-caching [11] and server dissemination [2] are server-driven techniques for caching of web objects. An origin server contacts caches, performs the tasks of locating objects, maintains concurrency and removes objects from caches. Push caching allows servers to set policies for objects, but requires the server to negotiate resources with caches and maintain state about which cache maintains copies of objects.

Active Cache [5] allows Java programs to be executed in the cache. The objective of this scheme is to be able to cache dynamic objects within caches. As a result, this scheme does not provide the same abilities to modify the behavior of the caching policies.

7 Summary

Caching systems are becoming common. However, caches are often limited in the policies that can be applied to cached web objects. This paper describes an infrastructure that provides customizable and dynamic policies for web objects.

We have designed an infrastructure that allows caches, web authors, and clients to customize policies. We have used this design to simulate a web cache and used traces from the DEC Squid cache and the NLANR cache infrastructure to assess dynamic and customizable caching policies. We have incorporated CacheL into Squid in order to assess the overhead of using an interpreted language to handle policies. Performance analysis shows that customized policies indeed allow caches to adapt to the requirements of clients, servers, and other caches. The overhead of adding customizable policies to a cache system is moderate.

Areas of future work include: efficiency, and a toolkit to provide several different policies already created to simplify the work of cache administrators and web authors.

Availability

More information on this project can be found by visiting the CacheL web page (`http://pdclab.cs.ucdavis.edu/qosweb/CacheL/`). This page includes a formal description of the CacheL grammar, script files used in running the experiments in this paper, source code for the DavisSim cache simulator, and patches that allow the embedding of our infrastructure within Squid.

Acknowledgements

We gratefully acknowledge Earl Barr, Brant Hashii, Peter Honeyman, Scott Malabarba, and the anonymous reviewers for their valuable suggestions. Thanks also to Tom Kroeger, Jeff Mogul, Carlos Maltzahm, and Digital Equipment Corporation for making the DEC logs available. We thank Duane Wessels at NLANR for making the NLANR cache logs available. The NLANR data used in some simulations were collected by the National Laboratory for Applied Networks Research under National Science Foundation grants NCR-9616602 and NCR-9521745.

References

[1] J. Fritz Barnes and Raju Pandey. CacheL: Language support for customizable caching policies. In *Fourth International WWW Caching Workshop*, San Diego, CA, USA, 31 March–2 April 1999.

[2] Azer Bestavros. WWW traffic reduction and load balancing through server-based caching. *IEEE Concurrency*, 5(1):56–67, January–March 1997.

[3] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. Web caching and Zipf-like distributions: Evidence and implications. In *Proceedings of IEEE INFOCOM '99, the Conference on Computer Communications*, New York, NY, USA, 21–25 March 1999.

[4] Pei Cao and Sandy Irani. Cost-aware WWW proxy caching algorithms. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, pages 193–206, Monterey, CA, USA, 8–11 December 1997.

[5] Pei Cao, Jin Zhang, and Kevin Beach. Active cache: Caching dynamic contents on the web.

In *Middleware '98. IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 373–88, Lake District, UK, 15–18 September 1998.

[6] Vincent Cate. Alex – a global filesystem. In *USENIX File Systems Workshop Proceedings*, pages 1–12. USENIX, May 1992.

[7] Steve J. Caughey, David B. Ingham, and Mark C. Little. Flexible open caching for the web. In *Proceedings Sixth International World-Wide Web Conference*, volume 29(8–13) of *Computer Networks and ISDN Systems*, pages 1007–17, Santa Clara, California, USA, 7–11 April 1997.

[8] Anawat Chankhunthod, Peter B. Danzig, Chuck Neerdaels, Michael F. Schwartz, and Kurt J. Worrell. A hierarchical Internet object cache. In *Proceedings of the USENIX 1996 Annual Technical Conference*, pages 153–63, 22–26 January 1996.

[9] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder. Summary cache: A scalable wide-area web cache sharing protocol. In *ACM SIGCOMM '98 Conference. Applications, Technologies, Architectures, and Protocols for Computer Communication*, volume 28(4) of *Computer Communication Review*, pages 254–65, Vancouver, BC, Canada, 2–4 September 1998.

[10] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. Hypertext transfer protocol – HTTP/1.1. RFC 2068, UC Irvine, Digital Equipment Corporation, M.I.T., January 1997.

[11] James S. Gwertzman and Margo Seltzer. The case for geographical push-caching. In *Proceedings Fifth Workshop on Hot Topics in Operating Systems (HotOS-V)*, pages 51–5, Orcas Island, WA, USA, 4–5 May 1995.

[12] Terence P. Kelly, Yee Man Chan, Sugih Jamin, and Jeffrey K. MacKie-Mason. Biased replacement policies for web caches: Differential quality-of-service and aggregate user value. In *Fourth International WWW Caching Workshop*, San Diego, CA, USA, 31 March–2 April 1999.

[13] Thomas Kistler and Hannes Marais. WebL – a programming language for the web. In *Seventh International World Wide Web Conference*, volume 30(1–7) of *Computer Networks and ISDN Systems*, pages 259–70, Brisbane, Qld., Australia, 14–18 April 1998.

[14] P. Krishnan and Binay Sugla. Utility of cooperating Web proxy caches. In *Seventh International World Wide Web Conference*, volume 30(1–7) of *Computer Networks and ISDN Systems*, pages 195–203, Brisbane, Qld., Australia, 14–18 April 1998.

[15] Ari Luotonen and Kevin Altis. World-wide web proxies. In *First International Conference on the World-Wide Web*, volume 27(2) of *Computer Networks and ISDN Systems*, pages 147–54. Elsevier Science BV, 1994. Available from: `http://www.cern.ch/PapersWWW94/luotonen.ps`.

[16] Mark Nottingham. Optimizing object freshness controls in web caches. In *Fourth International WWW Caching Workshop*, San Diego, CA, USA, 31 March–2 April 1999.

[17] Alex Rousskov and Duane Wessels. Cache digests. In *Third International WWW Caching Workshop*, Manchester, UK, 15–17 June 1998.

[18] NLANR hierarchical caching system usage statistics. `http://www.ircache.net/Cache/Statistics/`.

[19] Renu Tewari, Michael Dahlin, Harrick M. Vin, and Jonathan S. Kay. Beyond hierarchies: Design considerations for distributed caching on the internet. Technical Report TR98-04, The University of Texas at Austin, 1998.

[20] Duane Wessels. Squid internet object cache. `http://squid.nlanr.net/`.

[21] Duane Wessels. Intelligent caching for world-wide web objects. In *Proceedings INET '95*, Honolulu, HI, USA, 27–30 June 1995.

[22] Duane Wessels and K. Claffy. ICP and the squid web cache. *IEEE Journal on Selected Areas in Communication*, 16(3):345–357, April 1998. Available from: `http://ircache.nlanr.net/~wessels/Papers/`.

[23] Stephen Williams, Marc Abrams, Charles R. Standridge, Ghaleb Abdulla, and Edward A. Fox. Removal policies in network caches for world-wide web documents. In *ACM SIGCOMM '96 Conference Applications, Technologies, Architectures, and Protocols for Computer Communications*, volume 26(4) of *Computer Communications Review*, pages 293–305, Stanford, CA, USA, 26–30 August 1996. ACM.

[24] Roland P. Wooster and Marc Abrams. Proxy caching that estimates page load delays. In *Sixth International World Wide Web Conference*, volume 29(8-13) of *Computer Networks and ISDN Systems*, pages 977–86, Santa Clara, CA, USA, 7–11 April 1997. Elsevier.

[25] Lixia Zhang, Scott Michel, Khoi Nguyen, Adam Rosenstein, Sally Floyd, and Van Jacobson. Adaptive web caching: Towards a new caching architecture. In *Third International WWW Caching Workshop*, Manchester, UK, 15–17 June 1998.

Exploiting Result Equivalence in Caching Dynamic Web Content

Ben Smith, Anurag Acharya, Tao Yang, and Huican Zhu
Department of Computer Science, University of California
Santa Barbara, CA 93106

Abstract

Caching is currently the primary mechanism for reducing the latency as well as bandwidth requirements for delivering Web content. Numerous techniques and tools have been proposed, evaluated and successfully used for caching static content. Recent studies show that requests for dynamic web content also contain substantial locality for identical requests. In this paper, we classify locality in dynamic web content into three kinds: *identical* requests, *equivalent* requests, and *partially equivalent* requests. Equivalent requests are *not* identical to previous requests but result in generation of identical dynamic content. The documents generated for partially equivalent requests are *not* identical but can be used as temporary place holders for each other while the real document is being generated. We present a new protocol, which we refer to as *Dynamic Content Caching Protocol (DCCP)*, to allow individual content generating applications to exploit query semantics and specify how their results should be cached and/or delivered. We illustrate the usefulness of DCCP for several applications and evaluate its effectiveness using traces from the Alexandria Digital Library and NASA Kennedy Center as case studies.

1 Introduction

As many Web sites evolve to provide sophisticated e-commerce and personalized services, dynamic content generation becomes more popular. Using multi-processor or cluster-based servers can speedup resource-intensive dynamic content generation [6, 12, 24]. If successful, caching can provide significant additional benefit by reducing server load, end-to-end latency and bandwidth requirement. Numerous techniques and tools have been proposed, evaluated and deployed for caching static content. There has been recent interest in caching dynamic Web content as well [5, 7, 10, 14, 16]. Dynamic content has three forms of locality: *identical* requests, *equivalent* requests, and *partially equivalent* requests.

- **Identical requests:** These requests have identical URLs which result in the generation of the same content. Recent studies [14, 16] have shown that this locality can be successfully exploited for caching.
- **Equivalent requests:** The URLs of these requests are syntactically *different* but result in generation of *identical content.* For example, map servers (e.g. [13, 23]) frequently impose a grid on the coordinate space and return the same map when presented with any location within a given region in the grid. As another example, news servers may wish to return different front-pages for requests from users from different regions or different sites. In this case, requests from all clients in a group are equivalent.
- **Partially equivalent requests:** These requests are syntactically *different* but result in generation of content which can be used as a *temporary place-holder for each other.* For example, documents which are conditionally distilled to a lower-resolution version by a service (e.g., TranSend [12]) can be used as a place-holder for the originals. As another example, maps(e.g., MapQuest [13]) and aerial images (e.g., the TerraServer [17])

with more than a given degree of overlap could be used as placeholders for each other.

Exploiting similarity between dynamic documents has been shown to be beneficial for fast Web page delivery based on delta-encoding [4, 15, 18]. In this paper, we present a new protocol, which we refer to as the *Dynamic Content Caching Protocol (DCCP)*, to allow individual content generating applications (e.g. CGI applications, Java servlets etc) to specify equivalence between different GET-based requests they serve. This information can be used by web caches to exploit these additional forms of locality. Identical requests and equivalent requests can directly fulfilled using previously cached results. For partially equivalent requests, previously cached content can be immediately delivered as an approximate solution to the client while actual content is generated and delivered. This allows clients to browse partial or related or similar results promptly while waiting for more accurate information.

We have designed DCCP using the extension mechanism provided in HTTP 1.1 cache control directives [11]. This has several advantages. First, specifying result equivalence information as an extension of the HTTP header allows DCCP to be deployed incrementally — servers, proxies and individual content generating applications can be upgraded individually. In many cases, existing CGI scripts can be upgraded to generate these headers simply by using a short Perl or shell script as a wrapper. Second, since HTTP 1.1 specifies that headers *must* be propagated by compliant caches, DCCP directives can be expected to reach most caches. Finally, a declarative header-based specification of caching requirements allows value-added proxies such as *TranSend* [12] to compose server-provided directives and hints with their own. For example, the TranSend proxy provides a distillation service for heterogeneous clients. A distilled version of a web document is a lower-resolution (or a summary) version of the original and can be returned in response to a request for the original document. Since this functionality is provided by an intermediate node, the ability to augment and compose cache control specifications allows the results of such processing to be effectively cached by web caches closer to the client.

This paper is organized as follows. We first summarize the related work and then present a description of DCCP. We illustrate its utility for a variety of applications including a customizable news service, a location-dependent weather information service, server-side image maps, and a geographically indexed digital library. Finally, we discuss our current cache design and implementation for DCCP and evaluate the effectiveness of DCCP using real traces from the Alexandria Digital Library and NASA Kennedy Center and a synthetic trace for a weather forecast application.

2 Related Work

Exploiting similarities between dynamic documents generated by multiple invocations of the same web application was first proposed in [4, 15] for delta-encoding. This idea was explored further by Mogul et al. [18] as *query clustering*. They analyzed web proxy and packet-level traces to evaluate the extent of similarity between dynamic documents with the same "URL prefix" (usually identifying the generating application) but different suffixes (usually arguments to the generating application). They found that there is substantial locality in "URL prefixes" – the 100780 distinct dynamic content requests in their proxy trace were based on just 12004 prefixes. They also found that exploiting such similarities in delta-encoding can offer significant performance advantages. The DRP protocol [21] uses a checksum algorithm (e.g. MD5) to identify file equivalence and avoid unnecessary file download if checksum values do not change. These proposals place the responsibility of creating equivalence (or partial equivalence) classes on a caching agent and a server, with little or no support from the applications. DCCP, on the other hand, allows individual web applications to explicitly specify equivalence between different documents they generate, which can help delta-encoding in identifying similar documents.

The idea of partial result delivery has been

considered by Dingle et al. [9]. They propose that a caching agent could send previously cached data to a client so she/he could browse an old version while current data is being fetched. Optimistically transferring potentially out-of-date data to reduce end-to-end latency is also considered by Banga et al. [4] in delta-encoding. In this work, they propose sending the cached version of a dynamic document to the requesting agent (client or the next level proxy) while a delta is being fetched. DCCP lets application users *explicitly* define partial equivalence between the cached results and a new query and we can utilize the infrastructure of [9] or [4] to achieve partial result delivery in the implementation of a DCCP-aware caching agent.

Cao et al. [5] propose that a piece of Java code (a *cache-applet*) be attached to each dynamic document. This code is run whenever a request is received for a cached document. Cache applets can rewrite the cached document, return the cached document or direct the cache to refetch or regenerate the document. This approach, referred to as *Active Cache*, is extremely flexible and can be used to maintain consistency in an application-specific manner as well as dynamically modify existing documents. However, the flexibility of Active Cache comes with a price. It requires starting up a new Java process (virtual machine or compiled code) for every request. Keeping track of result equivalence requires creation and maintenance of a pattern-matching network. Since the cache-applet for each document is independent (for security reasons), cache-applets used to implement result-equivalence-based caching would need to save and restore its pattern-matching network to persistent storage. Depending on the access pattern, this can be a significant performance limitation. In contrast with the generality of Active Cache, DCCP is more restrictive. The trade-off is that this design simplifies implementation and provides opportunities for optimization. The limited scope and the declarative nature of the DCCP directives alleviates security concerns. Using a single global pattern-matching network allows processing for related patterns to be shared. Finally, since the structure and the function of the pattern-matching network is known to the cache, it can maintain portions of the network in memory.

Iyengar et al. [16, 7] propose that cache servers export an API that allows individual applications to explicitly cache and invalidate application-specific content. These techniques are developed for caching identical requests at original content providers and may not be feasible for proxy or client caches. These techniques can be integrated with DCCP when DCCP is deployed at server-sites.

In previous research, we have considered cooperative caching for dynamic web content on a cluster of servers [14]. The research focus is to study how clustered nodes collaborate with each other in terms of caching and the Time-To-Live method is used for maintaining result consistency. However, this work uses complete URLs as names for documents and has no notion of equivalent or partially equivalent requests.

Douglis et al. [10] propose that servers should make the structure of a dynamic document available to caches and that caches should construct the desired document on demand. The idea is that a dynamic document be specified as a static part, one or more dynamic parts and a macro-based template that combines them. The static parts and the template can be cached whereas the dynamic parts are fetched anew for every request.

3 The Dynamic Content Caching Protocol (DCCP)

The goal of DCCP is to let web applications exploit query semantics and specify equivalence or partial-equivalence between the dynamic documents they generate. DCCP has been defined using the extension mechanism provided in HTTP 1.1 cache control directives [11]. Figure 1 presents the proposed directives (see Section 4 for examples). Each document can contain one or more equivalence directives. Each equivalence directive specifies the set of requests that this document can be used to fulfill. The set of requests is specified using a pattern over the set of arguments embedded in the URL, client cookies, and the domain name and IP address

```
directive     := eq_result | sim_result
eq_result     := equivalent_result =condition
sim_result    := partial_result= condition
condition     := arg = pattern | (arg = pattern) op condition
arg           := __domain | __IP_address | field_name | cookie_name
cookie_name   := cookie:field_name
op            := "&&" | "||"
pattern       := range | regexp
range         := [number, number]
regexp        := "Perl syntax"
```

Figure 1: DCCP syntax for specifying equivalence directives. Field names are those argument names in a GET-based query.

of the machine from which the query is originated. There is no authentication requirement implied in these arguments – caches implementing DCCP are expected to provide best-effort values for these arguments.

The language used to specify the equivalence patterns provides equality for all arguments and range operators for numeric arguments. For example, "[0.0,100.0]" specifies any real number between and including 0 and 100. Patterns can be composed using logical operators ("&&" stands for "and" and "||" stands for "or"). The current version of DCCP also includes regular expressions over alphanumeric and special characters for string arguments as an experimental feature. We use Perl syntax [22] for regular expressions. Note that a directive need not specify values for all arguments. When a cache agent (e.g. a proxy server) receives a request, it extracts the application arguments from the URL, and the identity arguments from request header as well as the identity of the requesting machine. Note that DCCP has been designed for handling GET-based queries.

Since dynamic documents are generated on demand, they usually have greater consistency requirements than static documents. HTTP 1.1 provides several cache control directives to manage consistency. The current version of DCCP does not propose additional directives for this purpose. We believe more experience with dynamic documents is needed to determine the set of commonly used consistency requirements. If necessary, we could add directives to specify options such as polling periodically. The main problem with detailed consistency directives is that a proxy may not implement the directives which can result in incorrect results being delivered to clients. Thus our current focus is to support the class of applications which can benefit from DCCP with the existing HTTP 1.1 consistency mechanisms.

4 Applications

In this section, we illustrate how DCCP can be used to specify result equivalence for a variety of content generating applications.

Alexandria Digital Library: The Alexandria Digital Library (ADL) [3, 20] provides geo-spatially-referenced access to large classes of distributed library holdings. Current ADL collections contain more than 7 million entries of maps, satellite images, aerial photographs, text documents, and scientific data sets. Each entry has geographical extent represented as a minimum bounding rectangle in longitude and latitude and is spatially indexed. Queries used in the current version of ADL look like this:

```
GET /cgi-bin/draw_map?
lat=36.81818181&lon=-115.45454545&
ht=75.0&wd=180.0 HTTP/1.1.
```

Consider the scenario where the ADL server knows that the maps it is serving have a limited resolution. In that case, it can use the DCCP equivalence directives to define regions contained in the same map as equivalent. For example, for the above query, it could reply:

```
HTTP/1.1 200 OK
```

```
Server: Swala/1.8a
Content-type: text/html
Cache-control:
equivalent_result=''lat=[36,37]&&lon=
[-115,-116]&&ht=[74,76]&&wd=[179,181]''
```

The caching agent would associate this pattern with the map. If a subsequent request is received with arguments that match this pattern, the caching agent can return that result instead of calling the application and re-executing the request. We evaluate the impact of using DCCP for ADL dynamic content requests in Section 6.1.

Server-side image maps: Image maps [19] are used to provide intuitive and attractive labels for web links. Server-side image maps consist of images that is embedded in a web page. When a user clicks on a portion of the image, the browser submits a GET request to the server and which includes the screen coordinates as parameters. Client-side image maps can also be used to translate coordinates within simple image regions into appropriate HTML links. A client-site browser does such a translation and thus client-site image maps are preferable in terms of performance. However, server-side image maps are useful in cases where the image map is too complicated for a client-side image map or a server needs to process clicked screen coordinates. There are a number of high volume sites including `latimes.com`, and `washingtonpost.com` use the server-site image maps.

The typical number of regions on an image map is fairly small (e.g., from five to fifteen) while the possible number of coordinates is usually at least two orders of magnitude greater. Specifying result equivalence for all coordinates using DCCP can significantly improve the performance of server-side image maps. A cache without equivalence matching would be required to store all combinations of coordinates to generate a significant number of cache hits. With DCCP, the equivalent result already present in a cache can be identified without involving the server. We evaluate the impact of using DCCP for image maps in Section 6.2.

Weather Service: Weather servers use dynamically generated content to provide up-to-date weather information for the requested location. Typically location information, zip code or city name, is encoded in the request URL. Such requests, however, can be overly precise. Twenty zip codes might have the same weather information, but since their URLs are different, a cache with no support for result equivalence would require twenty cache entries to provide complete coverage. With DCCP, a weather server can specify that the same page should be returned for all requests in the same region. Figure 2 provides an illustration for zip-codes in and around the University of California, Santa Barbara. The request is `GET /cgi-bin/weather.cgi?zip=93101 HTTP/1.1`.

```
HTTP/1.1 200 OK
Server: Swala/1.8a
Content-type: text/html
Cache-control: equivalent_result=
''zip=93111||zip=93106||zip=93117''
```

Figure 2: DCCP directives for a weather application

Customized news services: There are several forms of customized news services. The simplest of these customize the content based on the location of the requesting clients. This is similar to the different editions that newspapers publish targeting specific regions. For such a scenario, all requests for a news page from the same geographical region (or Internet domain) would be equivalent. For example, all clients from `.uk` machines would be presented with United Kingdom specific news stories. The first time the page is accessed, the server must run the application to generate the customized web page; subsequent requests for the same page from machines with a `.uk` domain name can be serviced from the cache. The DCCP directive for this case is presented in Figure 3.

News services with greater degree of cus-

```
HTTP/1.1 200 OK
Server: Swala/1.8a
Content-type: text/html
Cache-control: equivalent_result=
''_domain=*.uk''
```

Figure 3: DCCP headers for a UK-specific web page

```
HTTP/1.1 200 OK
Server:  Swala/1.8a
Content-type:  text/html
Cache-control:  equivalent_result=''sports=yes&&clinton=yes&&movies=yes''
Cache-control:  partial_result=''sports=yes&&clinton=yes''
Cache-control:  partial_result=''sports=yes&&movies=yes''
Cache-control:  partial_result=''clinton=yes&&movies=yes''
```

Figure 4: DCCP directives for a user-profile-specific web page

tomization could use previously stored profiles to customize the news contents for individual users. Such a service can facilitate caching of its documents by marking documents generated for users with identical profiles as *equivalent* and documents generated for users with substantially overlapping profiles as *partially-equivalent*. Figure 4 provides an illustration.

5 Design of a DCCP-aware cache prototype

The primary challenge for the design and implementation of a DCCP-aware caching agent is to be able to efficiently maintain and use information about equivalence and partial equivalence of cached documents. We have developed a multi-stage pattern-matching network for this purpose based on the Swala cooperative web cache [14]. In this section, we describe the structure of this network and how the insertion and search operations are implemented.

In the DCCP model, each application specifies its result equivalence independently. As a result, a dynamic document generated by an application can be specified to be equivalent only to other documents generated by the same application. Accordingly, our prototype maintains a separate pattern-matching network for every application. Each application is identified by its *net-path-name*. The net-path-name for a URL consists of the longest prefix that contains no arguments. Our prototype uses a hash table (with net-path-names as keys) to quickly retrieve the pattern-matching network corresponding to a URL.

We discuss how we can build an efficient pattern matching network for each application with the same net-path-name as follows. In general a pattern matching rule can be transformed into a sequence of conjunctive pattern phrases connected by logical OR. For example, pattern `"map=USA&&lat=[34,37]&&lon=[-119,-117] ||map=USA&&city=foo"` contains two conjunctive phrases. Given a set of conjunctive phrases derived from the same rule or from different rules, we partition them into a set of clusters and each cluster has conjunctive phrases with the same argument names used in their exact match and range match patterns (discussed below). For example, two pattern phrases `''map=USA&&zip=93111''` and `''map=USA&&zip=93016''` are in the same pattern phrase cluster.

We discuss how each pattern cluster arranges its data structure for result matching. Given conjunctive pattern phrases within each cluster, we classify patterns into three parts in increasing degrees of complexity and searching within each cluster contains three stages:

- **Exact match**. All patterns are of form `argument=val`. These patterns can be hashed together. Searching within this cluster can start from the corresponding hash table so that only results satisfy those exact match patterns are candidates for further searching.

- **Range match**. All patterns are of form `argument=[val1, val2]`. Searching for a result under these conditions is similar to the problem of point intersection searching [2]: given a set of objects specified with multi-dimensional intervals, find an object that contains a query point. In our current implementation, we have used the R*tree data structure to group all such range match patterns together.

- **Complicated string match.** All patterns are of form `argument=regular_expression_pattern`. These patterns contain complicated string matching and so far we have not found a good way to organize a set of such patterns for efficient searching. Our current solution is to linearly scan such rules to find one result that matches these string matching patterns.

Thus the first stage of searching within each cluster is to use a hash table to narrow the searching space, i.e. find potential results which satisfy all exact match conditions. The second stage of searching is to use a point intersection data structure to further narrow the searching scope. The last stage of searching linearly scans all candidate results after the above two-stage filtering.

Each pattern network for an application with the same net-path-name consists of a layered graph with a single designated root node. Each child of this root node is a subgraph corresponding to each pattern cluster. Each cluster subgraph contains a hash table for exact match patterns and data structures for range match patterns. The cached dynamic documents constitute the leaves of this graph. A dynamic document is attached to a node in the graph if and only if the sequence of tests occurring on the path from the root to this node successfully satisfies all conditions specified by the DCCP rule of this document. Note that since each document can have multiple DCCP directives and since each directive can have multiple conjunctive-phrases, a cached document can be associated with multiple interior nodes in the pattern-matching network.

When a new DCCP-annotated dynamic document is inserted into the cache, the caching agent extracts the DCCP directives and the URL of the request that generated the document. It parses the URL to extract the *net-path-name* for the request and uses it to find the root of the associated pattern-matching network. It parses the DCCP directives and transforms them into a set of conjunctive-phrases and inserts each conjunctive-phrase into the network. Since cached dynamic documents may become stale and the rule for the same URL may change, the pattern network needs to be updated periodically.

When a new URL query arrives from a client, the prototype parses the URL to extract *net-path-name* for the request and uses it to find the root of the associated pattern-matching network. It then extracts the arguments from the URL and the cookies from the header. It also extracts the IP address of the requesting machine by querying the socket structure and does a reverse lookup to determine the domain name of the requesting machine. It then uses these values to perform tests against the pattern-matching network. Notice only some of query parameters required by the network are extracted for the matching purpose. If the match is successful, that is, the match operation reaches the node corresponding to a valid cached document, the corresponding document is deemed equivalent to the one requested by the client and is returned in response to the query. Notice that there may be multiple cache entries matching a new query and the system returns the first matchable cache entry it finds.

Our above optimization is targeted at rules that mainly use exact match or range patterns. It is possible that for complex patterns (particularly those making liberal use of regular expressions), trying to determine result equivalence could take an inordinately long time. To handle this case, we suggest that the system should limit the amount of time spent searching for a single request. Since result equivalence is only a hint, terminating the search early and fetching a new copy of the document from the server is safe.

For implementing progressive delivery of partially-equivalent results, we can use the `multipart/x-mixed-replace` MIME type, a mechanism available in the Netscape browser for providing continuously updatable web pages. The previous work has used that for delivering stale data [9].

6 Evaluation

To measure the space and time efficiency of equivalence matching, we performed tests for the following three applications: map delivery with boundary error tolerance, infor-

mation retrieval based on server-side image maps, and providing weather forecast based on zip code. We use real traces for the first two applications and a synthetic trace for weather forecast.

The goal of our evaluation was to estimate the performance improvement, if any, for DCCP-aware caching agents. For this, we mainly used two metrics: cache hit ratio and the volume of communication between the caching agent and the content provider (if this information is available). These metrics allow us to run repeatable experiments and to focus on the inherent performance improvements – independent of the network characteristics. In the ADL experiment, we also measured the end-to-end performance to demonstrate benefits in terms of response time reduction.

In these experiments, we ran a DCCP-aware proxy on a 248MHz, 128MB Sun Ultra-30 with Solaris 2.6. All the code was compiled with `gcc -O`.

6.1 Alexandria Digital Library web trace

In this experiment, we tried to estimate the performance improvement that can be achieved by using DCCP for spatial image searching and retrieval requests in the ADL system [3]. We have used an ADL user access trace for September and October 1997. This trace contains 69,337 requests and there are 28663 requests involving CGI. Of those dynamic requests, 21,545 were cgi-bin requests invoking one of two scripts: `draw_map` and `map-gif`. These scripts return portions of maps, based on the parameters which include central-latitude, central-longitude, height and width. Of those 21545 map requests, 7026 requests use an identical URL to access an ADL startup image with zero latitude and longitude and the remaining 14529 requests access maps with different parameter values. We used DCCP equivalence directives for those 14529 map requests to improve the cache hit ratio. We specified two map areas to be equivalent if the difference of above four parameters between two maps is within a specified error tolerance ratio (for example, 5% or 10%). Two requests are identical if the tolerance ratio is 0.

In this experiment, we set up a server at Rutgers University in New Jersey and ran both clients and the DCCP-enabled proxy at UCSB. The clients launch requests sequentially based on the trace to the proxy and the proxy can respond quickly if it has cached data since they are linked with a local network, or it fetches data from Rutgers. The round-trip latency between UCSB and Rutgers was around 82 milliseconds, measured by using "ping". The server at Rutgers does not have the ADL data installed, and it emulates the real ADL server at UCSB by executing each CGI request with processing time and result size same as what the UCSB ADL server would have. This experiment assumes that the server is fast enough to handle concurrent requests and it does not consider load impact among simultaneous requests. The experiment was conducted at midnight when Internet traffic is relatively stable and the average results are reported by running this experiment multiple times. There are minor variations among different runs; however, the deviation is fairly small.

The server processing time for each request was measured in 1997 [14]. At that time, we replayed the log against the ADL server to measure the response time and response size of each request. Considering today's machines can be 3-fold faster than the ADL server machine used in 1997, we have scaled down the request processing time accordingly in this experiment. The average processing time for all 28663 dynamic requests is about 0.5 seconds. The average processing time for all map requests (21545) is 169 ms while it is 162 ms for those 14529 requests whose results include the DCCP headers. The reason for the large difference in the average time between all dynamic requests and all map requests is that there are a certain number of ADL requests involving time-consuming spatial database searching.

Table 1 shows the benefits of equivalence caching for processing all dynamic ADL requests in this trace. The table shows that for exact matches, a 38.6% cache hit is achievable and this is because there are a large number of repeated requests with identical URLs. If the application can tolerate a 5% or 10% relative error, the hit percentage increases to 57.1% and 64.5% respectively. The total size

Error tolerance	0%	5%	10%
Number of cache hits	11074	16370	18482
Cache hit percentage	38.6%	57.1%	64.5%
Saved communication	48MB	65.2MB	75MB
Percent reduction in communication	41.5%	56.3%	64.8%
Ave. response time for a cache hit	4.8 ms	4.5 ms	4.4 ms
Ave. response time for a cache miss	773 ms	852 ms	958 ms
Ave. response time per request	476 ms	368 ms	340 ms

Table 1: Impact of result equivalence on cache performance when all dynamic requests are considered.

of data shipped in this trace is 115.6MB and the forth and fifth rows show the server-proxy communication volume saved due to the deployment of DCCP. The sixth row is the average request response time (from a client launches a request until it receives a result) for those requests whose results are fetched from the proxy cache at UCSB. The seventh row is the average request response time for those requests whose results are fetched from the Rutgers server due to cache misses. The eighth row is the average request response time for all dynamic requests. The results show that DCCP with tolerance 10% is 40% faster than only caching results of identical requests in terms of the average response time. If tolerance 5% is used instead, then it is 29.3% faster.

Now we focus on those map requests whose return results use the DCCP directives and there are 14529 such requests. Table 2 shows the benefits of equivalence caching and cost incurred due to the use of DCCP in processing those map requests. The table shows that for exact matches, only a 13.6% cache hit is achievable. If the application can tolerate a 5% or 10% relative error, the hit percentage increases to 49% and 62% respectively. The total size of data shipped for these requests is 61.3MB and the forth and fifth rows show the server-proxy communication volume saved due to the deployment of DCCP. The sixth row is the average request response time for all those map requests. The average cost for pattern matching network management and equivalence searching varies from 4.4 to 4.8 milliseconds per request while the memory usage for the entire pattern-matching network varies from 330KB to 900KB.

The performance improvement achieved by DCCP-aware caching agents comes, however, at the cost of additional memory to store the pattern-matching network. Even though the cost may not be large for many applications (e.g., see Table 2), it is conceivable that the memory requirement can grow rapidly if sufficient locality is not present in the request stream. This situation can be taken care of by imposing a bound on the amount of space used for the network. When the caching agent runs out of space for the pattern-matching network, it purged old entries using an LRU discipline. To understand the impact of such a restriction, we conducted additional experiments in which we limited the amount of memory that could be used for holding the pattern-matching network. Table 3 presents the results. The second row in Table 3 indicates percentage of space available compared to the situation when there is sufficient space. The third row shows the cache hit ratio under different space availability and error tolerance ratios. The result shows that this trace still has very good temporal locality and a high hit percentage can be maintained even when only half of the required space is available.

6.2 NASA Kennedy Space Center web trace

In this experiment, we tried to estimate the performance improvement that can be achieved by using DCCP for server-side image maps. For these experiments, we used a trace from the NASA Kennedy Space Center WWW server in Florida [1] to quantify the benefit of equivalence matching for image map type workloads. The trace contains

Error tolerance	0%	5%	10%
Number of cache hits	1975	7146	8959
Cache hit percentage	13.6%	49%	62%
Saved communication	7.9MB	26.MB	37MB
Percent reduction in communication	12.8%	46.6%	60.3%
Ave. response time per request	346 ms	217 ms	156 ms
Ave. cache search/insert overhead	4.8ms	4.5ms	4.4ms
Memory usage	900K	450K	330K

Table 2: Impact of result equivalence on cache performance for those DCCP-deployed requests.

Error tolerance	0%			5%			10%		
Percent space available	100%	80%	50%	100%	80%	50%	100%	80%	50%
Cache hit ratio	13.6%	12.8%	12.3%	49%	46%	45%	62%	58%	56%

Table 3: Impact of limited memory for the pattern-matching network.

1,569,898 requests, of which 20,925 are for the image map `countdown69`. We removed invalid requests, for a total of 20,774 requests. Image maps transmit the coordinates where the client clicked back to the server script, which translates the coordinates to a URL or returns a result. We have plotted the hit pattern in Figure 5. For each point that was clicked a hollow circle was plotted. The regions that have solid black areas are where multiple clicks occurred and correspond to buttons on the image map. Each button maps to the same URL, while points that do not fall into a button area result in an error message from the script. The image map `countdown69` has eleven buttons, as can be seen from the heavy concentration of dots in eleven regions. The scattering of dots outside of the buttons are clicks which do not correspond to buttons and result in an error message. We define each button as an equivalent region, as well as the error message as an equivalent region, resulting in twelve distinct regions on the image map.

Using standard URL matching results in a cache hit of 15,592 hits, and a 75.1% hit ratio. Using result equivalence for all points in the same region, the hit ratio increases to 99.9% or twelve cache misses, one for each region. The searching time and space cost is insignificant for this case because there are only twelve unique results and there are only twelve unique rules one per dynamic document.

It should be noted that for this trace, using client-site imagemaps can achieve the same goal as DCCP; nevertheless, this experiment demonstrates the effectiveness of DCCP if server-site images are deployed.

6.3 Weather forecast based on zip code

For this experiment, we generated a synthetic trace. We assume that the total number of zip codes is 99999 and this number is chosen based on the fact that a zip code in the U.S. has five digits. There are about 3143 counties in USA and for this test we assume that zip codes are randomly distributed among counties and each county in USA has the same weather condition within the same time period. In terms of access patterns of our synthetic trace, we assume a Zipf distribution [25, 8]. While we do not have evidence to support this assumption, we believe it is more realistic than a uniform distribution.

For a randomly generated trace with 500,000 requests, the average searching and network management cost is 0.12 milliseconds with a hit ratio of 99%. The reason for small time overhead is that there is a single application with one argument name (zip code). Hashing effectively indexes all zip codes, which allows for fast searching. The space usage is 1.6MB for indexing all 99999 zip codes in the pattern matching network because each rule enumerates all equivalent zip

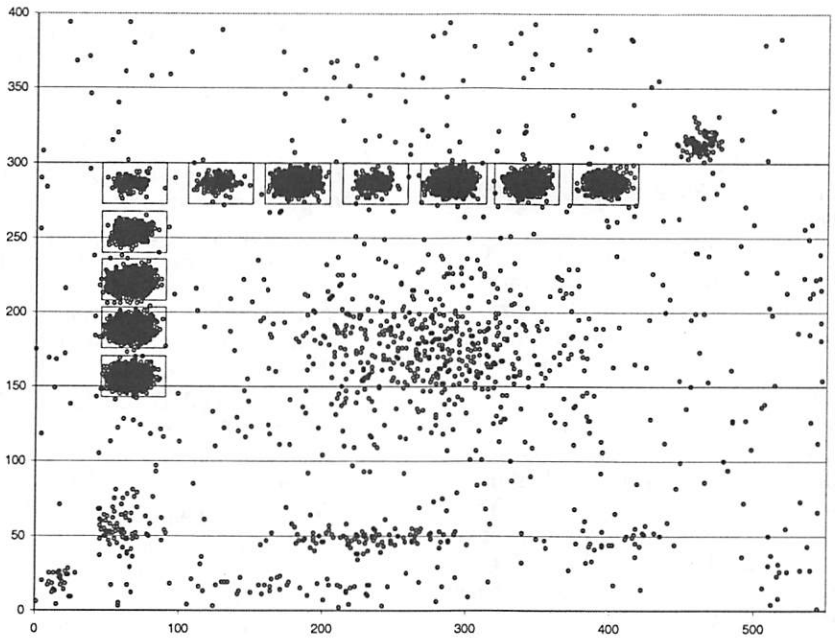


Figure 5: Hit pattern for `/cgi-bin/imagemap/countdown69` from the NASA trace. The hollow points correspond to coordinates where users clicked on the original image map. Boxes correspond to buttons on the original image.

codes for the corresponding result. Imposing a limit on space consumption does not degrade the cache hit ratio much because there are actually only 3143 unique results.

7 Concluding remarks

The primary contribution of this work is the DCCP protocol which allows a web application to specify result equivalence between the different documents (and groups of documents) it generates. This information can be used by proxies and other caching agents to speedup delivery of dynamic web content. We have illustrated the utility of this protocol for several applications and preliminary experiments indicate that the protocol is capable for achieving high cache hit ratios with fairly small space and time overhead.

DCCP can be extended to express more complicated patterns (e.g. inequality) and a few directives can be further added to the current DCCP for supporting basic features such as banner rotation and access logs, which are necessary for commercial Web sites [5]. Still, compared to Active Cache, the functionality of such a declarative protocol is more restrictive. The trade-off is that the declarative and lightweight nature of this caching protocol allows simplified security control and better efficiency.

We have not evaluated benefits of using our protocol for sending partially equivalent results and this issue needs to be addressed in the future. Under our current protocol implementation, searching equivalent results with complicated string matching rules can be time-consuming and caching may be less effective if imposing a searching time limit. One possibility is to restrict the use of string matching. Our current DCCP design is targeted at GET-based queries and there are a large number of dynamic requests which use POST-based queries. We plan to study the above issues after we gain more application experience with DCCP.

Acknowledgments. This work was supported in part by NSF CCR-9702640, IIS-9817432. We would like to thank Fred Douglis and anonymous referees for their detailed comments and thank K V Ravi Kanth and Ambuj Singh for providing their R* tree

code.

References

[1] ACM SIGCOMM. The Internet Traffic Archive. *http://ita.ee.lbl.gov/index.html*, Mar 1999.

[2] P. K. Agarwal and J. Erickson. Geometric range searching. In *Advances in Discrete and Computational Geometry (B. Chazelle, J. E. Goodman, and R. Pollack, editors)*, pages 1–56. Contemporary Mathematics 223, AMS Press, 1999.

[3] D. Andresen, L. Carver, R. Dolin, C. Fischer, J.Frew, M. Goodchild, O.Ibarra, R. Kothuri, M. Larsgaard, B. Manjunath, D. Nebert, J. Simpson, T. Smith, T. Yang, and Q. Zheng. The WWW Prototype of the Alexandria Digital Library. *Proceedings of ISDL'95: International Symposium on Digital Libraries*, Aug. 1995.

[4] G. Banga, F. Douglis, and M. Rabinovich. Optimistic deltas for WWW latency reduction. In *Proc. of USENIX'97*, pages 289–303, Jan. 1997.

[5] P. Cao, J. Zhang, and K. Beach. Active Cache: Caching dynamic contents on the web. In *Proc. of Middleware'98*, pages 373–388, 1998.

[6] J. Challenger, P. Dantzig, and A. Iyengar. A Scalable and Highly Available System for Serving Dynamic Data at Frequently Accessed Web Sites. In *Proc. of SC'98 (High Performance Networking and Computing. Formally known as SuperComputing)*, Nov. 1998.

[7] J. Challenger, A. Iyengar, and P. Dantzig. A scalable system for consistently caching dynamic web data. In *Proc. of IEEE INFOCOM'99*, Mar. 1999.

[8] M. E. Crovella and A. Bestavros. Self-Similarity in World Wide Web Traffic: Evidence and Possible Causes. In *Proc. of ACM SIGMETRICS Inter. Conf. on Measurement and Modeling of Computer Systems*, Apr. 1996.

[9] A. Dingle and T. Partl. Web cache coherence. In *Proc. of Fifth International WWW Conference*, May 1996.

[10] F. Douglis, A. Haro, and M. Rabinovich. HPP: HTML macro-preprocessing to support dynamic document caching. In *Proc. of USENIX Symp. on Internet Technologies and Systems*, pages 83–94, Dec. 1997.

[11] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. Network Working Group RFC 2616, June 1999.

[12] A. Fox, S. Gribble, Y. Chawathe, E. Brewer, and P. Gautheir. Cluster-based network services. In *Proc. of 16th ACM Symposium on Operating System Principles (SOSP'97)*, pages 78–91, Oct. 1997.

[13] GeoSystems Global Corporation. The MapQuest Home Page. *http://www.mapquest.com*, 1999.

[14] V. Holmedahl, B. Smith, and T. Yang. Cooperative caching of dynamic content on a distributed web server. In *Proc. of Seventh IEEE International Symposium on High Performance Distributed Computing (HPDC-7)*, pages 243–250, 1998.

[15] B. Housel and D. Lindquist. Webexpress: A system for optimizing web browsing in a wireless environment. In *Proc. of 2nd International Conf. on Mobile Computing and Networking*, pages 108–116, Nov. 1996.

[16] A. Iyengar and J. Challenger. Improving web server performance by caching dynamic data. In *Proc. of USENIX Symp. on Internet Technologies and Systems*, pages 49–60, Dec. 1997.

[17] Microsoft Corporation. The TerraServer Home Page. *http://www.terraserver.com*, Mar 1999.

[18] J. Mogul, F. Douglis, A. Feldmann, and B. Krishnamurthy. Potential benefits of delta-encoding and data compression for HTTP. In *Proc. of SIGCOMM'97*, pages 181–94, Sept. 1997.

[19] D. Raggett, A. L. Hors, and I. J. (Eds.). HTML 4.0 Specification. W3C Recommendation, http://www.w3.org/TR/REC-html40/, April 1998.

[20] T. Smith. A digital library for geographically referenced materials. *IEEE Computer*, 29(5):54–60, 1996.

[21] A. van Hoff, J. Giannandrea, M. Hapner, S. Carter, and M. Medin. The HTTP Distribution and Replication Protocol. Submitted to W3C. http://www.w3.org/TR/NOTE-drp, August 1997.

[22] L. Wall, T. Christiansen, and R. L. Schwartz. *Programming Perl.* O'Reilly & Associates, Inc., Sebastapol, California, second edition, Sep 1996.

[23] Xerox PARC. The Xerox Map Viewer. *http://pubweb.parc.xerox.com/map*, Mar 1999.

[24] H. Zhu, B. Smith, and T. Yang. Scheduling optimization for resource-intensive web requests on server clusters. In *Proc. of 11th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'99)*, pages 13–22, June 1999. http://www.cs.ucsb.edu/research/rcgi.

[25] G. K. Zipf. *Human Behavior and the Principle of Least Effort.* Addison-Wesley, Cambridge, MA, 1949.

Efficient Support for Content-Based Routing in Web Server Clusters

Chu-Sing Yang and Mon-Yen Luo

Department of Computer Science and Engineering
National Sun Yat-Sen University
Kaohsiung, Taiwan, R.O.C

Abstract

Clustered server architectures have been employed for many years on the Internet as a way to increase performance, reliability and scalability in the presence of the Internet's explosive growth. A routing mechanism for mapping requests to individual servers within cluster is at the heart of any server clustering techniques. In this paper, we first analyze the deficiencies of existing request-routing approaches. Based on these observations, we argue that the request routing mechanism in a cluster-based server should factor in the content of a request in making decisions. Thus, we designed and implemented a new mechanism to efficiently support content-aware routing in Web server clusters. With this mechanism, we also built in a number of sophisticated content-aware intelligence for making routing decision. Performance evaluation on a prototype implementation demonstrates substantial performance improvements over contemporary routing schemes. The proposed mechanism can also enable many new capabilities in cluster-based servers, such as sophisticated load balancing, differentiated service, special content deployment, session integrity, etc.

1. Introduction

The Internet, in particular the World Wide Web, has experienced explosive growth and continues to expand at an amazing paces [1]. This has resulted in heavy demands being placed on Internet servers and has raised great concerns in terms of performance, scalability and availability of Web services. A monolithic server hosting a service is usually not sufficient to handle these challenges. Cluster-based server architecture has proven [2,3,4] a successful and cost effective alternative to build a scalable, reliable, and high-performance Internet server system. In fact, popular Web sites are increasingly running Internet services on a cluster of servers (e.g., Alta Vista [5], Inktomi [6], Netscape [7]), and this trend is likely to accelerate.

An important issue that arises in such clustered architectures is the need to dispatch and route incoming requests to the server best suited to respond. Over the past few years, a considerable number of researchers and vendors have proposed methods for distributing the user requests in such clustered server. We classify these schemes based on where in the layer the request-distribution function is applied:

- **Client-side approach**
  First, request routing can be achieved through a Java applet that is downloaded and executed at client side (e.g., [8,9]). The applet provides service-specific customizations that forward each client request to an appropriate server.
- **DNS-based approach**
  When a client tries to request a document from a Web server, the client-side browser first has to construct a DNS query to resolve the mapping of server hostname (from URL) to IP address. The DNS-based approach [10,11,12] performs routing decisions at this name resolution level. The name server at the server side can be customized to resolve the same name to different IP addresses for the purpose of request distribution.
- **TCP connection routing**
  Once the IP address is resolved, the client will try to establish a TCP connection to the server with the designated IP address. A number of researchers [4,13,14,15] and vendors [16,17,18,19,20] have proposed a connection-routing frontend to route user requests at this stage. These front-end devices make parallel services on different servers appear as a virtual service on a single IP address.
- **HTTP redirection**
  After the TCP connection setup is completed, the client then issues the HTTP request. At this stage, the HTTP redirection approach (used in [21]) may use one special response code *URL Redirection* (defined in the HTTP protocol [22]) to perform request routing. This is achieved by returning the address of the selected server instead of returning the requested data, asking the client to create a second connection and resubmit its request to that

server.

We have previously designed and implemented a cluster-based framework [4,23] for building a scalable and highly available Internet server. From our experiences using this framework to build many large-scale Internet service sites, we found a number of important issues that cannot be effectively addressed by existing routing mechanisms. Examples include session (or transaction) integrity, sophisticated load balancing, quality of service, and content placement. We discuss these issues in section 2. Based on these observations, we argue that in many cases the routing mechanism in a cluster-based server should factor in the content (e.g., URL or service type) of a request in making decisions. We refer to this concept as "content-based routing." We propose a new mechanism to efficiently support content-based routing in Web server clusters. The design of the proposed mechanism is presented in section 3. We implemented the mechanism as a specialized software component called *content-aware distributor*, which is a software module for kernel-level extension. We present the key portions of the implementation in section 4. Section 5 presents the results of performance evaluation of the prototype system. The results demonstrate substantial performance improvements over contemporary routing schemes. We compare our system with related work in section 6, and then present the conclusion in section 7.

2. Issues Ignored by Existing Routing Schemes

Existing approaches for routing requests to individual servers within cluster have typically concentrated on \user transparency, load distribution, and scalability. From our experience of running Internet services on a cluster architecture, we argue that a number of other issues are very important as well:

Session Integrity: Currently, the HTTP protocol is stateless, i.e., a Web server fulfils each request independently without relating that request to previous or subsequent requests. However, there are many situations (e.g., electronic commerce) where it is essential to maintain state information from previous interactions between a client and a server. For example, such state might contain the contents of an electronic "shopping cart" (a purchase list in a shopping mall site) or a list of results from a search request. When the user checks out of the shop, or asks for the next 25 items from a search, the state information from the previous request is needed. A number of schemes are employed (e.g., cookies [24] or hidden variables within HTML FORM [25]) or proposed [26] for handling state on the Web. Unfortunately, these methods might not be processed properly in a cluster-based server. If the routing mechanism do not examine the content of each request in selecting a server, it is possible a request belonging to a session is dispatched to the wrong server. This could limit the usefulness of cluster architecture because the state concept is an increasingly critical part of Web behavior for e-commerce, Web-oriented database, and other dynamic transaction applications.

Sophisticated Load Balancing: A server cluster requires some sort of load-balancing mechanism for directing requests in a way that utilizes the cluster resources evenly and efficiently. In current Web sites, the service type of incoming requests can be as varied as static Web pages, dynamic content generated by CGI scripts, or multimedia data such as streaming audio or video. The service time and the amount of resources consumed by each request vary widely and depend on several factors. For example, a request for executing a CGI script normally requires much more computing resources compared to static file retrieval requests [27]. This heterogeneity in request often causes skewed utilization of the server cluster. As a result, a more sophisticated load-balancing mechanism based on the service type of each request is essential. The load-balancing capability provided in many existing systems is still limited because they do not consider the service type of each request. Typical load-balancing strategies used in these systems include Round Robin [11, 14], Weighted Round Robin [17], Least Connections [18], Weighted Least Connections [17], and Monitored Server Load [17,18]. We previously implemented these mechanisms in our Web server cluster, and observed that these techniques still cannot deliver satisfactory performance when the workload characteristics of each request change significantly. Others have made similar observations [28, 29].

Differentiated Services: The Web continues to evolve from its initial role as a provider of read-only access to static documentation-based information, and is becoming a platform for supporting complex services. However, most current Web servers, both cluster-based and monolithic, provide service in a best-effort manner that does not differentiate between the requirements of different requests. This approach does not work well with advanced services and potential future needs. Different services may have different requirements for quality of service. If the routing schemes do not take the service type of each request into consideration, it is difficult to enforce priority policies and to provide the desired quality of service. Otherwise, not all content are

equally important to the client and service provider. However, we notice that requests for popular pages have the tendency to overwhelm the requests for other critical pages (such as product list or shopping-related pages). As a result, enterprises and service providers (e.g., Web content hosting service providers) may want to exert explicit control over resource-consumption policies, to provide differentiated quality of service due to the variety of content.

Content Deployment: Given a cluster-based server, how to place and manage content in such a distributed system is an important and challenging issue. Because the incoming requests may be distributed to any server node in the cluster, each participant server must have the same capability for responding to requests for any portion of resource that the Web site provides. Typically, this requirement can be achieved by a shared network file system or full replication of all content on each server. However, neither of these two schemes is a satisfactory solution. The networked file system approach will suffer from the single-point-of-failure problem and increase user perceived latency by accessing data over the network file system. Full replication of content is expensive in terms of space utilization, and will not work for some Web services (e.g., a Web service using a commercial database). Thus, there are times when either to reduce space requirements or for performance reasons, a Web site administrator wants to place different functions or content on different sets of servers. In cases where this is done for performance, specific machines can be dedicated to a particular task (e.g., executing CGI scripts) for performance optimization. To support such flexible content deployment, the routing scheme must be content-aware so that it can know which server hosts the requested content.

These observations lead to the inevitable conclusion that in many cases the routing mechanism in a cluster-based server should factor in the content of requests in making decisions. However, this requirement cannot be effectively addressed by existing routing mechanism. A routing mechanism generally has to collect some information about server's state to perform routing decisions. When request routing is performed at the client side, the status information that is collected remotely may be stale, resulting in bad routing decisions. For example, servers that appear to be underutilized may quickly become overloaded because everyone sends their requests to those machines until new load information is propagated. Both DNS-based and connection-router approaches are content-blind, because they determine the target server before the client sends out the HTTP request. HTTP redirection might be used for content-aware routing. However, we do not prefer HTTP redirection because this mechanism is quite heavy-weight. Not only does it necessitate the use of one additional connection, which introduces an extra round-trip latency, but also the routing decision is performed at the application level and uses the expensive TCP protocol as the transport layer. Motivated by these observations, we propose a new mechanism, called a content-aware distributor, to effectively support content-based routing.

3. The Content-Aware Distributor

The major challenge in designing the content-aware mechanism is the connection-oriented semantics of TCP protocol that the Web service is based on. Whenever a client tries to issue a request to the Web server, it must first establish a TCP connection to the server. As a result, the dispatcher node has to establish a TCP connection with the client so that it can examine the subsequent HTTP request to perform content-based routing. When the routing decision has been made, relaying the HTTP request (or migrating the established connection) from the dispatcher to the selected server becomes a challenging problem. In particular, such a mechanism should be transparent to users and efficient enough to not render the dispatcher a bottleneck.

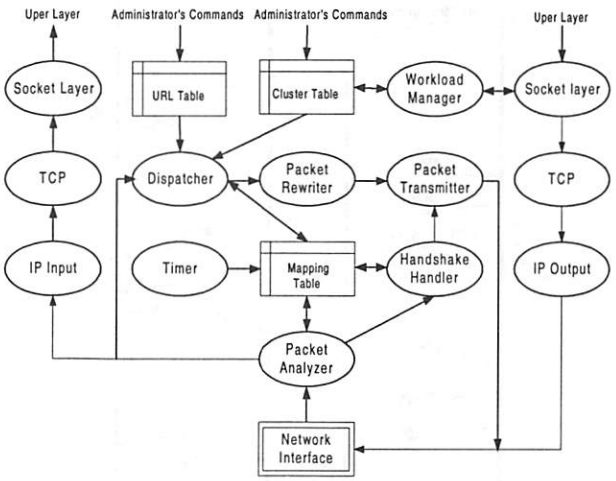


Figure 1. Functional Overview of Content-Aware Distributor

Figure 1 shows our design of the content-aware distributor. The operation of the content-aware distributor (distributor for short) is largely based on the following three data structures that hold vital information for routing a request: the cluster table, the mapping table and the URL table. Each entry in the

cluster table represents one participant server in the cluster. The most important fields of the cluster-table entry include the IP address and MAC address of one participant server, a timer for failure detection, the load index (a measure of current load), and the available connection list (which will be described later).

The distributor also maintains and manipulates the mapping table for directing a request to the selected server. Each entry in this table is indexed by the source IP address and source port of the client, and also includes the IP address and port number of selected server, a state indication, TCP state information for the client and the selected server, and a timestamp. The URL table holds information (e.g., location of the document, document sizes, type, priority, etc.) that helps the dispatcher to make routing decisions.

To explain the operation of the distributor, we look at the sequence of events when a client requests a document from a Web server, and then show how the distributor operations fit into the packet exchange (see Figure 2).

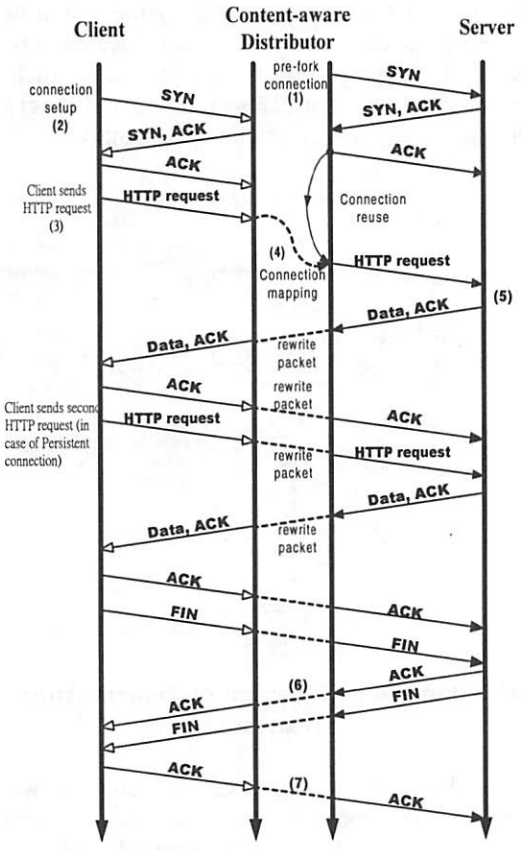


Figure 2. Operation of Content-Aware Distributor

3.1 Connection Setup

Whenever a client tries to initiate an HTTP request (position 2 in figure 2), the client-side browser first opens a TCP connection, resulting in an exchange of SYN packets as part of TCP's three-way handshake procedure [30].

For each incoming packet, the packet analyzer inspects the information embedded in the IP and TCP header. If this packet is not destined for the Web service, the packet is passed up to the normal protocol stack. Otherwise, the mapping table is consulted to find the corresponding mapping information. If no information about this packet is found, the packet analyzer checks the SYN flag in the TCP header. If the SYN flag is set, representing the arrival of a new TCP connection, the handshake-handler module is invoked. The handshake-handler module first creates an entry in the mapping table for this connection and sets the status of this entry to SYN_RCVD. Then it handshakes with the client to complete the TCP connection setup. When connection establishment is completed, the status is changed to ESTABLISHED and the TCP state information (e.g., sequence number, ACK number, etc) is recorded in the mapping-table entry.

3.2 Transmitting the HTTP Request

After the TCP connection setup is completed, the client sends packets conveying the HTTP request for the specific content it is looking for. Once such a packet arrives, if the packet analyzer can find a corresponding entry in the mapping table and the status of this entry is ESTABLISHED, then the dispatcher module is invoked. The dispatcher parses the URL of this request, and then looks up the URL table and cluster table to select the server that possesses the requested content and is least loaded.

The cluster table is maintained by the workload manager which performs the following three functions. First, it estimates the current load on each server node to help the dispatcher make the optimal decision. Second, it monitors the health of the server nodes to bring them transparently in and out of service. Third, it pre-forks (i.e. creates a socket and establishes a TCP connection) a number of TCP connections to each server (position 1 in figure 2). These pre-forked connections are kept long-lived.

Once the dispatcher selects a target server, it also chooses an idle pre-forked connection from the available connection list. Then the dispatcher stores related information about the selected connection in the mapping table and changes the state to ACTIVE, which

will bind the user connection to the pre-forked connection (position 4 in figure 2). The first objective behind such a design is to avoid the overhead of initiating a new connection to the selected server every time an HTTP request is made. Otherwise, it can increase user perceived performance by avoiding multiple slow-starts [31], because multiple successive requests conveyed by different user connections will be served by the same long-lived connection. We will demonstrate the advantages of this approach in section 5.

After the connection binding is determined, the packet-rewriter function is invoked to change the packet's IP and TCP headers for relaying the packet from the user connection to the pre-forked connection. After the packet's header is modified, the packet transmitter function is invoked for directing the packet to the selected node. The packet transmitter will find the network interface on which this packet should be forwarded. Then it constructs a network frame for transmission to the indicated interface. The forwarded frame has the MAC address of the network interface of the selected server.

3.3 Subsequent Data

When the designated server receives the request, it parses the URL to determine the content requested. It then fulfills the request and transmits the response to the client. The distributor also intercepts these response packets and performs the reverse packet modification so that the client can transparently receive and recognize these packets.

The distributor also handles the persistent connections suggested by HTTP 1.1 [22]. HTTP 1.1's persistent connections use one TCP connection to carry multiple HTTP requests, thereby reducing server load and client perceived latency [32]. Our mechanism inherently supports persistent connections. For HTTP 1.0 requests (non-persistent connections), the distributor can reuse the pre-forked connection to carry these requests, which will avoid extra TCP 3-way handshakes and multiple slow-starts. Otherwise, the distributor splits multiple HTTP 1.1 requests within a persistent connection into single requests. If these requests belong to the same session, the dispatcher routes them to the server assigned to the same session. Else, it schedules the individual HTTP 1.1 requests to different servers based on content of these requests.

3.4 Connection Termination

If the packet analyzer receives a FIN packet, which indicates that one host participating in this TCP connection intends to close this session, the state of corresponding entry in the mapping table will be changed to FIN_RECEIVED. Later, when the peer responds with an ACK packet to the FIN packet (position 6 in figure 2), the state is changed to HALF_CLOSED. When both FIN and ACK from two participating hosts in this TCP session arrive (position 7 in figure 2), the state is changed to CLOSED, and then the entry associated with this connection will be deleted after 2MSL (Maximum Segment lifetime, which is used to handle wandering packets in the Internet [30]).

If the client fails or loses connectivity with its server, there may be no FIN or ACK to arrive as anticipated above. This will result in accumulation of stale entries in the mapping table. To solve this problem, we maintain a timestamp for each TCP connection in the mapping table, which will timestamp the connection records each time a packet flows through them. The timer function periodically checks and deletes expired entries, defined as those that have been idle for a configurable amount of time. When one entry is deleted, the bound pre-forked connection is released to the available connection list, and it remains open for next client.

4. Implementation

For both efficiency and elegance, we implemented the proposed mechanism as a loadable module for Linux kernel. The kernel loadable module is a software component that can be dynamically loaded into kernel for the purpose of kernel extension. The distributor module inserts itself between the network interface (NIC) driver and the TCP/IP stack. By processing packets at this level, preventing them from traversing the protocol stacks, the distributor can analyze and route packets quickly. This effectively reduces the overhead of the distributor. To route incoming Web requests, all packets relating to Web service should go through the content-aware distributor. To achieve this, we can load the content-aware distributor module into the default router of the clustered Web servers.

Due to space limitations, we only present the key portions of the implementation. In particular, we focus on describing how the proposed mechanism can be implemented efficiently without imposing a significant amount of overhead. Because the ideas and mechanisms adopted in the content-aware distributor are generic, they should be applicable to other systems (e.g., BSD or Windows NT) as well. Some functions of the content-

aware distributor are derived from our previous work [4,23]. Examples include failure detection and handling, workload evaluation and balancing, and a primary-backup mechanism for circumventing the single-point-of-failure problem. We do not describe these functions in this paper. For a detailed description, see [4, 23].

4.1 Hash Search

For all incoming packets, the mapping table must be consulted to find the corresponding mapping information. As a result, the process of locating the entry (binding information) relevant to each incoming packet can become a significant performance factor. To facilitate efficient access, insertions, and deletions to corresponding entry, we implemented the table as hash tables [33, 34] with doubly linked lists as collision resolution chains. Otherwise, we also implemented a mechanism to cache recently accessed entries, which is a proven [35, 36] technique for demultiplexing speedup.

The dispatcher also relies on information provided by URL table to make the routing decision. The URL table is implemented as a multi-level hash table, in which each level corresponds to a level in the content tree. Each item of content in the Web site has a record corresponding to it in the URL table. Each entry includes information for making an intelligent decision. Examples include locations of the content, document sizes, type, priority, etc. The URL table is initialized and computed upon initialization of the distributor by scanning and parsing the content tree.

4.2 Packet Rewriting

To relay a packet from the user connection to the pre-forked connection correctly (vice versa), the packet rewriter must modify the packet's IP and TCP header before forwarding:

- Change source and destination IP address to that of the pre-forked connection.
- Update checksum in the IP header.
- Change source and destination port number (if necessary) to that of the pre-forked connection.
- Map the packet sequence number from the user connection to the pre-forked connection.
- Map the packet ACK number from the user connection to the pre-forked connection.
- Map TCP options (e.g., timestamp, window scale, and maximum sequence size) as needed.
- Update the TCP header checksum.

We have designed and implemented an algorithm to map the sequence number, ACK number, and TCP options from the user connection to the pre-forked connection. We find the idea behind the algorithm is similar to the TCP-splicing technique proposed by [37]. For the sake of brevity and space limitations, we omitted the description of this algorithm. The reader can see [37] for further information. In addition, changing these headers requires that the IP checksum and the TCP checksums be updated. Re-computing the entire checksum is expensive [38]. We implemented a method (described in [39]) to incrementally update the checksum.

4.3 Connection Management

The rationale of pre-forking long-lived connections (described in section 3.2) is to reduce overhead and latency of establishing a second connection. However, an idle open TCP connection incurs a system-resource cost (e.g., socket, buffer space, and PCB entry [40]). Thus, we designed and implemented a connection-management mechanism in the workload manager to strike a good balance between the benefits and costs of maintaining open connections. The connection-management module uses two parameters to control the number of pre-forked persistent connections: maximum number of connections (MAX for short) and minimum number of connections (MIN for short).

MAX defines the threshold between system trashing and maximum optimization of system resources. The workload manager periodically queries each server node for its current load to determine the value of MAX. The connection management mechanism keeps a fixed limit (MIN) on the number of available persistent connections. When the total connections (used and unused) reach MAX, it stops pre-forking connections. To date we use a trivial fixed length timeout policy for closing idle connections. An adaptive timeout mechanism may be more appropriate in some cases. This is an area of further research and beyond the scope of this paper.

4.4 Dispatcher

The dispatcher is responsible for making routing decisions based on content-aware intelligence. Up to now, we have implemented the following content-aware intelligence:

- **Affinity-Based Routing**
  A request routing function can reduce retrieval latency not just by balancing loads and maintaining

low overhead but also through a third mechanism: affinity-based routing. An important factor to consider is that the latency for the server to serve a request from the disk is far longer than the latency to serve the request from the memory cache. With the content-aware mechanism, it is possible to direct requests for a given item of content to the same server so that the server can respond more quickly, due to an improved hit rate in the memory cache. In other words, requests are always dispatched to servers that already have data cached in main memory. Due to significant locality presented in user's references [41], the overall performance can benefit quite strongly from such a design. However, naively affinity-based routing achieves higher cache hit rates at the possible expense of load balancing. A number of algorithms have been proposed to overcome this problem. Examples include [42, 43]; our implementation is based on [42].

- **Dispersing Content Placement**

The content-aware mechanism enables a new content placement scheme, which allows the administrator to partition the content by hosting portions on different servers. The content partition may be governed by type (e.g., static HTML pages, CGI scripts, multimedia files, etc.) or by some other policy (e.g., priority). We implement some administration functions for administrator to configure the URL table to deploy the content tree. The new content-placement scheme offers several advantages over the traditional schemes. First, compared to full replication, this scheme yields better resource utilization and scalability while also avoiding the overheads associated with NFS approach. According to the traffic characterization of a modern Web site [44], large files (up to 64 KB) make up only 0.3% of the content but consume 53.9% of the required storage space. In addition, these large files receive only 0.1% of all client requests. Thus, full replication of these files is not cost-effective. We can just place these files on some nodes in the cluster. Second, we can place different content on different server optimized to address the requirements imposed by various data types. For example, we can place multimedia content on a server optimized for stringent real-time requirements. Finally, it provides the Web site manager with greater flexibility in selecting the optimum cost/performance configuration for their Web server. This is an important feature for cluster-based servers. Cluster-based servers tend to be heterogeneous because they generally grow incrementally as needed. In such a heterogeneous environment, some nodes may not be powerful enough to support an entire service, but can probably support some components of the service. The content-aware mechanism enables the administrator to place different content on different nodes according to their capability, which can maximize server investment returns.

- **Content Segregation**

From our previous experiences, we noticed that resource-intensive dynamic content processing (e.g. a complex database query) has the tendency to slow down the requests for static content, resulting in longer response times for these short requests. As a result, we suggest that the dynamic content should be segregated from the static content on different nodes. Another reason supporting the idea of content segregation is that the dynamic content is generally non-cacheable, so the affinity-based routing strategy is not directly applicable to it. We implemented a Weighted Least Connections algorithm to dispatch the requests for dynamic content. For requests to static content, the dispatcher performs routing decision with the affinity-based routing mechanism.

4.5 Future Directions

Because the function of the distributor is well defined and modularized, it could be easily extended or customized to meet unique system requirements. In the future, we will enhance the capability of the content–aware distributor by adding more content-aware intelligence. We will further explore more sophisticated load-balancing techniques based on the desired content of each request. Further, we will design and implement mechanisms for supporting session integrity so that the state shared by multiple requests in a session not be disrupted or delivered to wrong node. Finally, we also plan to design and implement some mechanisms to enable differentiated service in a clustered server. For example, it is increasingly common for a web site offering service both to paying subscribers and the public. Requests by paying customers should be given preferential treatment over those of nonpaying ones. The content-aware mechanism can include admission-control functions to reject some low-priority requests in highly overloaded circumstances, preventing the server from thrashing and guaranteeing the responsiveness of high-priority requests (e.g., requests by paying users, requests to critical pages such as home page or order form).

5. Performance Evaluation

In this section, we present our performance evaluation of the proposed mechanism. We used WebBench 3.0 [45] as the benchmark to evaluate the performance. The

server cluster consists of the following machines connecting through 100Mbps Fast Ethernet: one Pentium 150MHZ machine (with 128M RAM) running the Linux operating system (with a modified kernel) serves as the distributor, and six Pentium 150MHZ machines (with 64M RAM) serve as back-end servers. Some of the back-end servers run Windows NT with IIS, and the others run Linux with Apache. The reason for such a configuration is that we want to show that the servers clustered by our mechanism can be heterogeneous. We used 16 Pentium 150MHZ machines (with 64M RAM), which serve as WebBench clients.

5.1 Overhead of Content-Aware Routing

To quantify the overhead of the content-aware distributor, we measure and compare the response time of a Web request (for static content) traveling either across or not across a distributor for a variety of sizes. The overhead is defined to be the difference in response time between the two situations. We utilize WebBench to load the two configurations with many concurrent requests for the same file and then measure the response time. The results are given in Table 1. The values in the "baseline" are the response time for sending the request directly to the Web server. The additional overhead introduced by the content-aware distributor is acceptable. The latency via our content-aware distributor is about 4-10% slower than direct access. Notice that this experiment is performed over a local area network, where high-speed connections are the norm. The overhead would be insignificant when compared with the latency over wide-area networks with lower bandwidth. [46].

File size (Kb)	2K	8K	32K
Overhead (ms)	0.224	0.325	0.468
Latency(baseline)	5.124 ms	5.484 ms	7.42 ms
Percentage	4.8%	5.9%	6.3%
File size (Kb)	64K	256K	1024K
Overhead (ms)	0.767	2.325	9.524
Latency(baseline)	10.386 ms	26.973 ms	93.454 ms
Percentage	7.4%	8.6%	10.2%

Table 1 Overhead of Content-Aware Routing

We conduct another experiment to confirm that our design can effectively reduce the overhead. We first disable the pre-forking mechanism. That is, once the routing decision has been made, then the distributor creates a second TCP connection to the selected server. When the second connection has been setup, the distributor then starts to relay packets to the selected node. Under such a situation, the overhead increased by 89 μsecs. More importantly, such a change obliges the distributor to allocate memory buffer to queue the pending packets, increasing the burden of the distributor.

5.2 Benefits of the Proposed System

To demonstrate the benefits of the affinity-based routing mechanism, we first conducted experiments in the following environments: (1) 6 server nodes clustered by the content-aware distributor; (2) 6 server nodes clustered by the NAT (Network Address Translator [11,12]) router. The NAT router is the implementation in our previous work [4]. In the NAT router, we implemented a "Weighted Round-Robin" mechanism for load distribution, which is content-blind.

We used the WebBench benchmark with its standard static test suites to generate the client workload in these experiments. This standard static test suites define the workload WebBench uses to simulate traffic at actual Web sites. The workload is representative because it was created by examining log files supplied by many real Web sites, such as ZDNet, the Internet Movie Database, Microsoft, and USA Today [45]. Figure 3 shows the results in terms of throughput. It clearly shows that the server cluster with the content-aware distributor consistently achieved a greater throughput than the NAT cluster. The result also indicates that the additional overhead introduced by content-aware distributor can be compensated by sophisticated content intelligence.

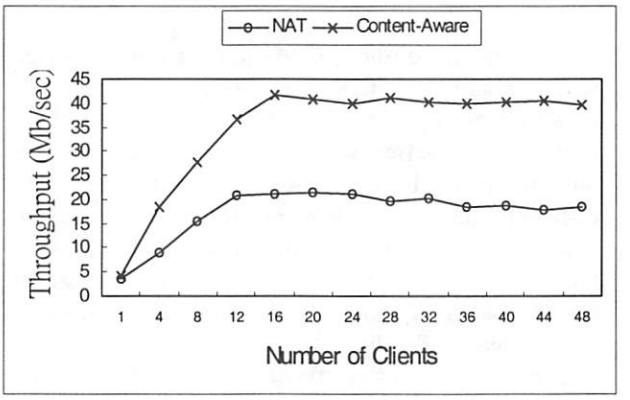


Figure 3. Benefit of Proposed System (Throughput)

We conducted another experiment to quantify the performance benefits of content-aware routing incorporated with content segregation. We used WebBench with a dynamic test suite (a mix of 20 percent CGI requests and 80 percent static requests) for

this evaluation. We measured the performance under the same two environments described above. In the NAT cluster, the content sets are also replicated in each participating node. For the content-aware cluster, we separated dynamic content and static content on different servers.

Figure 4 shows the results. The results show that the throughput achieved with content-aware routing outperforms that of Weighted Round-Robin. In the content-aware router with content segregation, the average CGI request and average static request increased by 27 percent and 36 percent respectively. The reason for this higher performance is because the content segregation prevents short Web requests (e.g., request for static content) from being delayed by long running request.

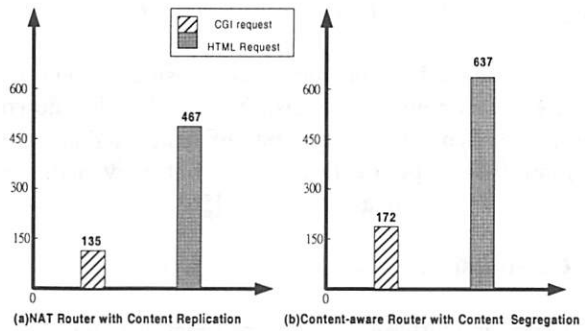


Figure 4. Benefit of Content Segregation (Average request/second)

5.3 Lessons Learned

Examining the results of our performance evaluation, it is noteworthy that the overhead increases with document size. This is because our mechanism performs some data-touching operations such as forwarding the packet and computing checksums. In our current implementation, packets from the server to the client must pass back through the distributor, and header modification is performed on each packet. If packets can go directly from servers to the client without having to pass through the distributor, the overhead can be further decreased. This is particularly beneficial while the amount of data sent from the server to the client is significantly larger than the amount of data sent from the client to the server.

Thus, we implemented a module that can be loaded into the kernel of the back-end server. The module can change the outgoing packets so that they can go directly to client. However, many Web site builders may not prefer this approach because it requires modification of the back-end server's kernel. In contrast, our previous implementation can be applied to any existing Web site without modification of backend server, but it pays a performance penalty. We suggest that the administrator can use the first approach for those nodes executing kernels that cannot be modified, and apply the second approach to those nodes that we can modify for performance gain.

6. Comparison

In this section, we compare our mechanism with the existing methods and related work. We discuss the advantages and disadvantages of various mechanisms.

NCSA [10,11] first proposed a clustering technique that uses the DNS-based approach, which has the advantages of low overhead and ease of implementation. The main weakness of this approach is that the name-to-IP-address mapping can be cached by multiple levels within the hierarchy of DNS service [46], bypassing address resolution. Thus, all requests behind a particular name server will be directly sent to the same server node until the cached address expires. This will lead to significant load imbalance, which has been quantified in [48,49], and several researches have attempted to overcome the problem [50,51,52]. However, another problem is that internal changes of the clustered server will propagate slowly through the Internet, due to address caching. That is, if one server node fails or is removed temporally for maintenance, a number of clients may continue trying to access the failed server using the cached address. In contrast, our mechanism is much faster than the domain name service in detecting failures and responding to it.

The connection-routing approach [13-20] can achieve fine-grained control in incoming requests compared to the DNS-based approach. However, since all connections to a server cluster have to pass through the front-end, it is a single point of failure and may become a bottleneck at high loads. Additionally, because they do not look into the content of requests, they are incapable of using more sophisticated load balancing policies. The request distribution strategies used in them generally are variations of weighted round-robin (or weighted least connections).

The HTTP redirection approach has the potential to support content-aware routing. The main disadvantage of this approach is that a request may require two or more connections for getting the desired service, which will increase the response time and network traffic. Furthermore, every request is initially addressed to the "scheduler" server, which also creates a single point of

failure and the potential for a bottleneck due to servicing redirects.

The major advantage of our approach is that it can effectively support content-aware routing. In contrast, none of the schemes described above actually support content-aware routing, which will limit the usefulness of their server clusters. We have shown that content-aware routing is essential in many cases. Our implementation can be easily extended or customized to add more content-aware intelligence.

Recently, some commercial start-up companies have also proposed similar idea of content-based routing. Examples include Arrowpoint [53], Resonate [54], and HydraWeb [55]. Several features of our works differentiate it from these commercial products. First, we give a detailed description of implementation and performance evaluation. We believe that some of our experience may be helpful in this area. Second, our approach reuses the pre-forked connection and seamlessly relays packets from the client-side connection to a pre-forked connection. We have demonstrated that such a design can decrease latency. In contrast, the product by Resonate defers opening the back-end TCP packet until it sees the content of this request, and then decides which server to redirect (and open/SYN) the packet to. It then encapsulates the entire IP packet, as it was sent from the client, and sends it to the selected server. Third, we have discussed the content-aware issue more widespread (e.g., session integrity, affinity-based scheduling, differentiated service) and have given some viable solutions to the problem.

Authors in [43] proposed a routing strategy called LARD for content-based request distribution. With LARD, the front-end may distribute incoming requests in a manner that achieves high locality in the back-ends' main memory caches as well as load balancing. Their work is focused on algorithm design and simulation for validation. Although they proposed a protocol to migrate established TCP connections, they did not describe how to implement their approach. In this aspect, our work is complementary to theirs. We provide a general and implementable solution for supporting content-aware routing and propose optimization techniques to make the mechanism efficient. In addition, their protocol requires modifications to the TCP implementation of backend server's kernel.

The major drawback of our approach is the extra processing overhead. However, this overhead can be minimized by efficient implementation, which is discussed in section 4. The performance-evaluation results show that the overhead is not substantial. In addition, we also demonstrate that the additional overhead can be compensated for by sophisticated content intelligence (e.g., direct requests to the best-fit node).

A second possible drawback is limited scalability. Due to the additional processing overhead of examining the content of each request, the distributor may become the impediment to scaling the server. To eliminate this problem, we can combine the proposed mechanism with a DNS-based scheme to further improve scalability. That is, a number of distributors can be used when the single distributor becomes a performance bottleneck, and the DNS-based approach can be used to map different clients to different distributors.

Finally, the distributor represents a single point of failure, i.e., failure of the distributor will bring down the entire Web server. Questions of fault tolerance are beyond the scope of this paper, but they will be discussed in detail in a future work [56]

7. Conclusion

Web server clustering is an important technique for constructing a high-performance, reliable and scalable Web server. However, the deficiencies of existing request-routing mechanisms limit the usefulness of the cluster-based architecture. In this paper, we analyze these problems and then argue that the request-routing mechanism should factor in the content of a request in making decisions. We designed and implemented an efficient mechanism to support content-aware routing. The performance evaluation results show that the additional overhead introduced by the mechanism is insignificant. With this mechanism, we also built in a number of sophisticated content-aware intelligence for making routing decisions. Performance evaluation on a prototype implementation demonstrates substantial performance improvements over state-of-the-art routing schemes that use only load information to distribute requests.

Our content-aware mechanism can enable many new capabilities in cluster-based server, such as sophisticated load balancing, differentiated service, special content deployment, session integrity, etc. While the usefulness of most existing Web server clustering schemes were constrained by lack of content-aware intelligence, our mechanism can enable many new services such as electronic commerce, database searching, and Web content hosting on cluster-based

servers. This will dramatically increase the usefulness of the Web-server clustering technique.

Acknowledgements:

We would like to thank our anonymous reviewers for their valuable comments. We also wish to express our gratitude to Dr. Geoff Kuenning for his proof-reading and making a number of helpful suggestions.
This work was supported by the National Science Council, R.O.C., under contract no. NSC 88-2213-E-110-022.

References:

[1] Internet Weather Report (IWR). Available at http://www.mids.org.

[2] T. E. Anderson, D. E. Culler, and D. A. Patterson, "A case for NOW (Networks of Workstations)," IEEE Micro, 15(1):54-64, February 1995.

[3] A. Fox, S. Gribble, Y. Chawathe and E. A. Brewer, "Cluster-based scalable network services," Proceedings of SOSP '97, pp. 78-91. St. Malo, France, October 1997.

[4] C. S. Yang, M. Y. Luo, "Design and implementation of a environment for building scalable and highly available web server," Proceedings of 1998 International Symposium on Internet Technology, pp. 124-131 April 29- May 1, 1998.

[5] Digital Equipment Corporation. About Alta Vista. http://www.altavista.com/av/content/about.htm.

[6] Inktomi Corporation. The Inktomi Technology Behind HotBot, a White Paper. http://www.inktomi.com

[7] D. Mosedale, W. Foss, and R. McCool, "Lessons learned: administering netscape's Internet site," IEEE Internet Computing, 1(2): 28-35, 1997.

[8] C. Yoshikawa, B. Chun, P. Eastham, A. Vahdat, T. Anderson, and D. Culler, "Using smart clients to build scalable services," Proceedings of the 1997 USENIX Annual Technical Conference, pp. 105-117, January 6-10, 1997.

[9] Y. M. Wang, P. Y. Chung, C. M. Lin, and Y. Huang, "HAWA: a client-side approach to high-availability web access," Proceedings of the Sixth International World Wide Web Conference, April 1997.

[10] E.D. Katz, M. Butler, and R. McGrath, "A scalable HTTP server: the NCSA prototype," Computer Networks and ISDN Systems, 27:155-164, 1994.

[11] R. McGrath T. Kwan and D. Reed, "NCSA's World Wide Web server: design and performance," IEEE Computer, November 1995.

[12] T. Brisco, "DNS support for load balancing," RFC 1794, http://www.internic.net/ds/

[13] H. Y. Yeom, J. Ha, and I. Kim, "IP multiplexing by transparent port-address translator" Proceedings of the 10th USENIX System Administration Conference (LISA X) Sep. 29 – Oct. 4, 1996 Chicago, IL, USA.

[14] E. Anderson, D. Patterson, and E. Brewer, "The magicrouter, an application of fast packet interposing," http://HTTP.CS.Berkeley.EDU/~eanders/projects/magicrouter/osdi96-mr-submission.ps

[15] O. Damani, P. Chung, Y. Huang, C. Kintala, and Y. Wang, "ONE-IP: techniques for hosting a service on a cluster of machines," Computer Networks and ISDN Systems, 29, 1997.

[16] D. Dias, W. Kish, R. Mukherjee, and R. Tewari, " A scalable and highly available Web server," Proceedings of the COMPCON'96, pp.85-92, Santa Clara, CA, February 1996

[17] IBM Corporation. The IBM Interactive Network Dispatcher 1998. http://www.ics.raleigh.ibm.com/netdispatch

[18] CISCO. Local Director. http://www.cisco.com/

[19] F5Labs. BigIP. http://www.f5.com/

[20] Foundry Networks. ServerIron Server Load Balancing Switch. http://www.foundrynet.com, 1998.

[21] D. Andresen, T. Yang, V. Holmedahl, and O.H. Ibarra, "Sweb: towards a scalable world wide web server on multicomputers." Proceedings of the 10th International Parallel Processing Symposium.

[22] T. Berners-Lee, R. Fielding, H. Frystyk, J. Gettys, J. C. Mogul, Hypertext Transfer Protocol – HTTP/1.1, http://www.w3.org/Protocols/

[23] C. S. Yang and M. Y. Luo, "Design and implementation of an administration system for distributed web server", Proceedings of the 12th USENIX Systems Administration Conference (LISA'98), pp. 131-140, Boston, Massachusetts, December 6-11, 1998.

[24] Netscape Communications Corporation. Client Side State - HTTP Cookies. http://www.netscape.com/newsref/std/cookie spec.html.

[25] D. Raggett, A. Le Hors, and I. Jacobs, HTML 4.0 Specification, W3C Working Draft, (1997).

[26] D. Kristol and L. Montulli, "HTTP state management mechanism," RFC 2109, Feb. 1997.

[27] A. Iyengar, E. MacNair and T. Nguyen, "An analysis of web server performance" Proceedings of the IEEE 1997 Global Telecommunications Conference (GLOBECOM '97), Phoenix, AZ, November 1997.

[28] H. Zhu, T. Yang, Q. Zheng, D. Watson, O.H. Ibarra and T. Smith, "Adaptive load sharing for clustered digital library servers," Seventh IEEE International Symposium on High Performance Distributed Computing (HPDC-7), Chicago, July 1998.

[29] H. Zhu, B. Smith and T. Yang, "A scheduling framework for Web server clusters with intensive dynamic content processing" Technical report, university of California, Available at http://www.cs.ucsb.edu/~hczhu/publications/rcgi.ps

[30] G. Wright and W. R. Stevens, *TCP/IP Illustrated, Volume1*, Addison-Wesley, Reading, May 1994

[31] J. Heidemann, "Performance interactions between P-HTTP and TCP implementations." ACM Computer Communication Review, 27 2, 65-73, April, 1997.

[32] J. C. Mogul, "The case for persistent-connection HTTP," Proceedings of the SIGCOMM'95, pages 299-313. ACM, August 1995.

[33] R. Jain, "A comparison of hashing schemes for address lookup in computer networks," IEEE Transactions on Communications, October 1992.

[34] P. E. McKenney and Ken F. Dove, "Efficient demultiplexing of incoming TCP packets," ACM SIGCOMM 92, August 1992.

[35] J. C. Mogul, "Network locality at the scale of processes," ACM Transactions on Computer Systems, May 1992.

[36] C. Partridge and S. Pink, "A faster UDP," IEEE/ACM Transactions on Networking, July 1993.

[37] D. Maltz and P. Bhagwat. "TCP splicing for application layer proxy performance," IBM Technical Report RC-21139, March 1998.

[38] B. Braden, D. Borman, and C. Partridge, RFC 1071: Computing the Internet checksum, Sept. 1988.

[39] A. Rijsinghani, RFC 1624: Computation of the Internet checksum via incremental update, May 1994.

[40] J. C. Mogul, "Operating system support for busy Internet servers," Proceedings of the Fifth Workshop on Hot Topics in Operating Systems (HotOS-V),Orcas Island, WA, May 1995

[41] S. D. Gribble and E. A. Brewer, "System design issues for Internet middleware services: deductions from a large client trace," Proceedings of the 1st USENIX Symposium on Internet Technologies and Systems, December 8-11, 1997, Monterey, California, USA

[42] D. Thaler and C. Ravishankar, "Using name-based mappings to increase hit rates," IEEE/ACM Transactions on Networking, Vol. 6, No. 1, February 1998.

[43] V. Pai, M. Aron, M. Svendsen, G. Banga, P. Druschel, W. Zwaenepoel, and E. Nahum, "Locality-aware request distribution in cluster-based network servers," Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems, October 1998.

[44] M. Arlitt, T. Jin, "Workload Characterization of the 1998 World Cup Web site" Hewlett-Packard Technical Report, February 1999.

[45] WebBench, http://www.zdbop.com

[46] P. Barford and M. E. Crovella, "Measuring Web performance in the wide area," Performance Evaluation Review, August 1999. Also available at http://www.cs.bu.edu/faculty/crovella/paper-archive/wawm-per.ps

[47] P. Mockapetris, Domain Names - Concepts and Facilities, Information Sciences Institute, University of Southern California, November 1987. RFC 1034.

[48] J. C. Mogul, "Network behavior of a busy Web server and its clients." Technical report, Digital Western Research Lab, October 1995. WRL Research Report 95/5.

[49] M. Colajanmi, P. S. Yu, and D. M. Dias, "Scheduling algorithms for distributed web servers," Proceedings of the 17th International Conference on Distributed Computing Systems, pages 169-176, Baltimore, MD, May 1997.

[50] M. Colajanmi and P. S. Yu, "Adaptive TTL schemes for load balancing of distributed web servers," ACM SIGMETRICS Performance Evaluation Review, 25(2):36-42, September 1997.

[51] M. Colajanni, P. S. Yu, D. M. Dias, "Analysis of task assignment policies in scalable distributed Web-server system," IEEE Transaction on Parallel and Distributed Systems, vol. 9, no. 6, June 1998.

[52] V. Cardellini, M. Colajanni, P. S. Yu, "Efficient state estimators for load control policies in scalable Web-server cluster," Proceedings of IEEE 22nd Int. Computer Software and Application Conference (COMPSAC'98), Vienna, Austria, Aug. 1998.

[53] Arrowpoint. http://www.arrowpoint.com/

[54] Resonate, http://www.resonate.com.

[55] HydraWeb. http://www.hydraweb.com/

[56] C. S. Yang, M. Y. Luo and C. W. Tseng, "Fault-tolerance Web Server", Submit for publication.

Rapid Reverse DNS Lookups for Web Servers

William LeFebvre
Group Sys Consulting
Alpharetta, GA 30022
WNL@GroupSys.com

Ken Craig
CNN Internet Technologies
Atlanta, GA 30348
Ken.Craig@CNN.com

Abstract

When a web server wants to learn the domain name of one of its clients, it must perform a lookup in the Domain Name System's "reverse domain", *in-addr.arpa*. These lookups can take time and may have an adverse impact on the web server's response to its clients. Rapid DNS is an intermediate client/server system that operates between a web server and a DNS server. It provides caching of the results and, more importantly, limits web server lookups to the data contained in the cache. This provides a significant improvement in response time for situations in which knowledge of the hostname is not critical to the web server's operation. The Rapid DNS system was implemented for use in the web farm that serves the collection of Cable News Network (CNN) sites. Its design is presented, along with measurements of its performance in the CNN environment.

1 Introduction

When a client connects to a server, the only information about the client that is available to the server is the client's IP address. In order to learn more about the client, the server must perform a DNS lookup in the *in-addr.arpa* domain, called a *reverse lookup*, to translate a client's IP address in to a name [9]. On widely accessed web servers, a high percentage of the reverse lookups will involve name servers from distant networks. Consequently, these lookups can take a long time.

Most high traffic web sites cannot afford to wait for the completion of reverse lookups, as the delay in processing these lookups would have a detrimental impact on the site's response time. Therefore, client tracking is limited to just IP addresses. Any desired demographic information must be generated off-line. Real-time determination of a visitor's origins is not a reasonable possibility due to the time required to perform a reverse lookup.

The CNN web farm supports approximately 50 web servers which provide content for sites known as *cnn.com*, *cnnfn.com*, *cnnsi.com*, and many others. A single web server in this farm can see as many as 20,000 hits per minute. The farm was designed from the beginning for simplicity, reliability and speed in order to support a web site that is the most heavily trafficked news site on the internet. In addition to serving over 20 million page views daily, the web farm must be able to withstand traffic spikes that are three times what is experienced on a normal day.

The farm consists of smaller, distributed servers which can be easily replaced or re-purposed. The web server software is primarily off-the-shelf, and additional software, in the form of web-server plug-ins, must not introduce significant latency to routine requests. Specialized functions are generally distributed off of the main-line servers to protect the basic service. A relatively homogenous environment simplifies the process of re-purposing hardware when the need arises.

Off-line DNS processing has provided the web farm team with useful information for analysis, but offers little benefit to advertisers; an important consideration for an advertising-supported web site. Domain based ad targeting was one of the most highly requested features to be added to our advertising capabilities due to its supposed simplicity and universal acceptance of accuracy. While architecturally simple, implementation of such a capability at scale requires a different solution.

As beneficial as such targeting may be, protecting the reliability of the primary web serving functions always takes precedence. This is basis for the two primary design requirements of any additions to the CNN Web Farm, including Rapid DNS; high performance during normal operations and graceful degradation of service under excessive traffic loads.

Even though it may be possible for a well-configured name server to handle several hundred requests per sec-

ond, there will still be a problem with latency. The root name servers are expected to sustain a minimum response rate of 1,200 queries per second, but they also disable recursion on all requests [7]. Server load may be an issue, but far more critical is the need to provide a quick response. A reverse lookup will need to consult name servers throughout the world and can take several seconds to complete. A server that receives over 300 requests per second cannot afford to have each request delayed by a recursive lookup.

In section 2 we present related work. Section 3 lays the foundational premise on which the entire system is built. Section 4 describes details of the Rapid DNS client, server, and protocol. Section 5 discusses management of the cache used by the server. Section 6 explains the specialized way in which IP addresses are queued internally for processing by DNS. Section 7 discusses the use of negative caching in Rapid DNS. Section 8 presents performance results for a variety of configurations. Section 9 discusses the results, and section 10 looks to the future.

2 Related Work

Surprisingly little work has been done in this area. A search of the published body of work has revealed no documented efforts to provide rapid reverse lookups for web servers.

Work has been done to utilize DNS for load balancing requests across multiple web servers ([1], [6]) and for integrating DNS lookups with HTTP redirection to achieve load balancing [3]. Brisco discussed the viability of using DNS as a general load balancing tool [2] and Schemers developed a Perl tool for tailoring DNS answers based on measured load [11]. A study of name server traffic on the NSFNet was conducted by Danzig, Obracza and Kumar [4], and they observed (among other things) that negative caching of DNS responses by servers would have little impact on the reduction of DNS packets across a wide-area network.

3 Premise: "I don't know" is acceptable

The desire for instant domain name information on the CNN Web farm drove us to implement a mechanism for rapid resolution. The foundational philosophy for Rapid DNS is that the answer "I don't know" is acceptable. If a name is not readily available when requested, then Rapid DNS is free to answer "I don't know". The web server then proceeds as if it never performed the lookup. This philosophy allows the implementation to uncouple queries from the actual resolution of the name

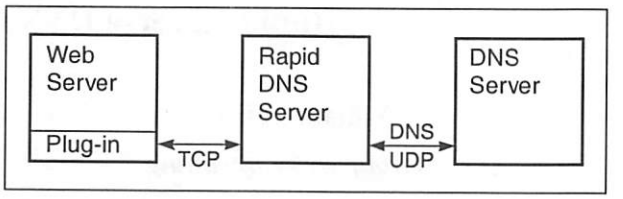


Figure 1: Inter-Server Relationship

Is it reasonable to accept non-answers for this sort of query? For our purposes the answer is yes. The host name information is needed for two separate purposes. First, we want to be able to produce summary traffic information correlated by top level domain. This allows us to calculate zone demographic information on our audience: "35% of our traffic was from educational sites" for example. Before the deployment of Rapid DNS we performed name resolution off-line: traffic logs carried the IP address and the translation to name was done *en masse* overnight after the logs were extracted from the web server. In this situation an address that cannot be translated in to a name is just placed in an "unknown" category. Clearly the same can be done with "I don't know" answers. Although this will adversely affect the demographic results, it will not hamper the operation of the web server itself.

The second purpose for domain names is to drive the selection of advertisements presented on the page. If the web server knows the domain name of the client it has the option to choose an advertisement specifically targeted for an audience group implied by the domain. If the name information is missing then the advertisement selection can just draw from a generic pool of advertisements. We miss a chance to target an ad, but are still able to operate.

There is a situation in which the absence of a domain name will have an impact on the server's operation: name-based authentication. If the server determines accessibility based on the domain name then the answer "I don't know" is not acceptable. A user who has legitimate access may not receive that access if the name lookup service can't provide an answer. The CNN web servers do not use name-based authentication, so this was not a concern for our service.

4 Design of the Rapid DNS

Rapid DNS is implemented as an intermediate service placed between the web server and the DNS name server (see Figure 1). A plug-in or module in the web server acts as the Rapid DNS client and is invoked as part of normal page handling. The Rapid DNS server can be run on any host accessible to the web server via

TCP, reading and responding to client requests. The Rapid DNS server is the only component that actually issues DNS queries. For the remainder of this paper, unless otherwise indicated, we will use the term "server" to refer to a Rapid DNS server and "client" to refer to a Rapid DNS client, even though that client may be part of a web server.

4.1 Client

The client we have developed for the Netscape Enterprise server uses a single persistent Rapid DNS connection to handle all requests in a given process, even though that process may have many threads handling HTTP requests

The design relies heavily on the multi-threaded capabilities of the Netscape server provided through its application programming interface (API) [10]. At initialization time, the Rapid DNS client starts several background threads. One thread, the writer, dispatches requests to servers. Additional threads, the readers, are created to read and process the servers' responses (one thread per server).

The Netscape server creates threads for handling HTTP requests. When a request arrives it is dispatched to an idle thread. While processing the HTTP request, the Rapid DNS plug-in function will be invoked and will place the peer's IP address on a central queue for processing. The writer thread takes a request off the queue, dispatches it and moves it to a pending queue that is specific to the server used. As a reader thread processes a response, it is matched up with the corresponding request in the pending queue. One response may serve to fill more than one request.

Request processing in the Netscape server is performed in 6 phases: authorization, name translation, path checking and modification, object typing, request servicing, and logging. Server plug-in functions can be invoked during any phase. Information is passed in to the functions using parameter blocks, or *pblocks*. These are hash tables that store name/value pairs and provide for easy lookup and modification. All information about the request is stored in a set of pblocks, and plug-in functions affect processing of the request by modifying these pblocks.

The Netscape API provides a function, *session_maxdns*, that retrieves the domain name of the HTTP client host. This function also stores the information in a pblock: specifically in the session client pblock using the parameter name *dns*. Subsequent calls to *session_maxdns* will use the information found there rather than perform the DNS lookup again. If DNS lookups are turned off in the Netscape server's configuration, then *session_maxdns* will not send any DNS queries, but it will still look in the pblock for the name.

The Rapid DNS plug-in, *rdns_lookup*, takes advantage of this behavior. When an answer arrives from the server, *rdns_lookup* will place it in the client pblock as the parameter *dns*. Subsequent calls to *session_maxdns* will never use DNS directly but rely exclusively on the information in the pblock, even if DNS lookups are turned off in the Netscape configuration. If the answer received from Rapid DNS is "I don't know" then no information is placed in the pblock, and calls to *session_maxdns* will return NULL. Beyond the change to the pblock, *rdns_lookup* affects no aspect of processing a request. As a consequence, it can be used in any phase of request processing. The following C statement illustrates how the name information is placed in the pblock:

```
pblock_nvinsert("dns", name, sn->client);
```

The *rdns_lookup* function must be invoked before any plug-ins that may need to utilize its information, such as advertisement scheduling software. CNN chose to use a configuration that invokes the client as the very last object type function, so that it is run immediately before entering the request servicing phase.

The writer thread will always dispatch requests to the server with the shortest pending queue. This policy automatically compensates for malfunctioning or abnormally slow servers. If a server fails, its persistent connection is severed and the client code will stop dispatching requests to the server. Finally, a watcher thread monitors all the queues to ensure that none of the requests get stuck.

This client design provides good scalability for web servers. In fact, we have Rapid DNS deployed across more than 60 web servers, and they all use the same trio of servers.

4.2 Server

The server also uses a threaded design. A thread is created to handle each client connection, and other service threads handle various maintenance tasks and communications with DNS servers. The server is logically separated in to two components: the front end and the back end. The front end handles the task of providing answers to clients, while communications with DNS servers is completely isolated to the back end. This total decou-

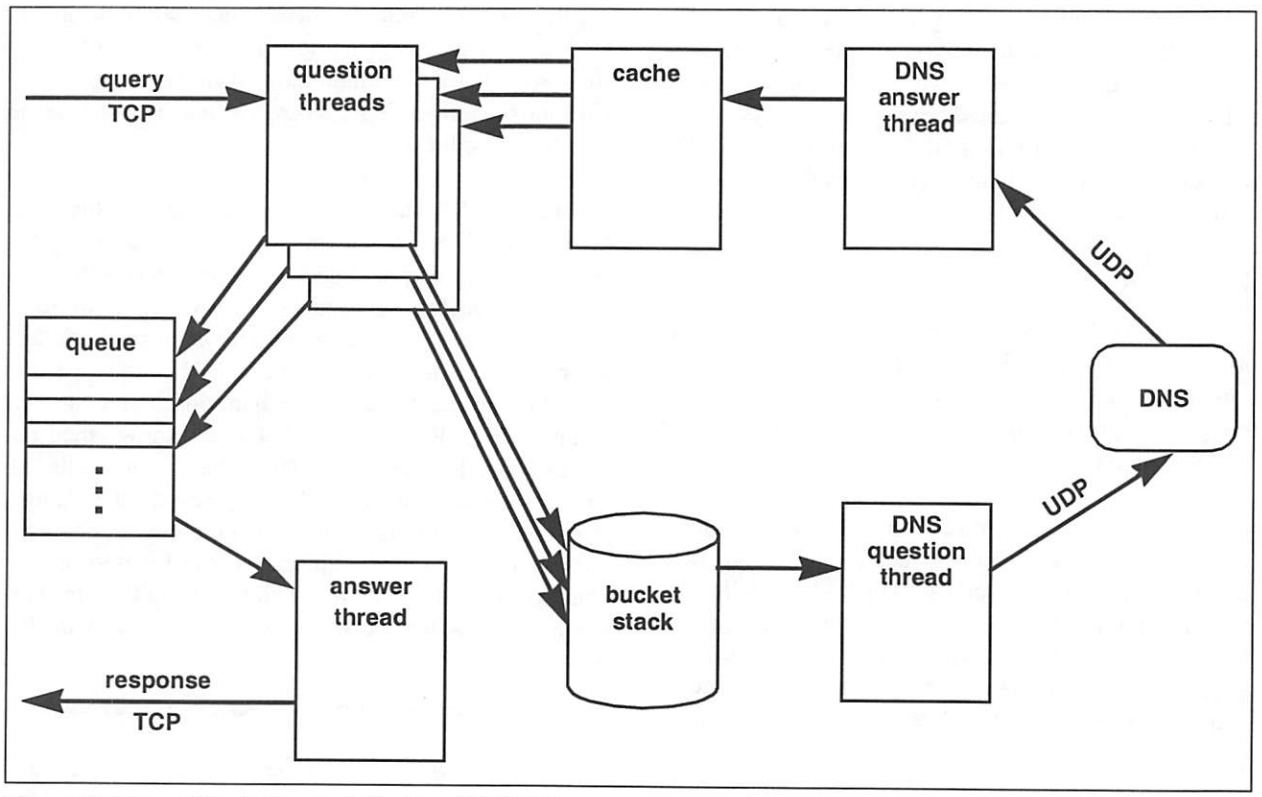


Figure 2: Flow of Data through the Rapid DNS Server

pling gives Rapid DNS the ability to provide quick results without waiting on answers from DNS servers.

The front and back ends are tied together with a cache and a stack. The cache, fed by the back end, contains all the DNS answers that the server has received. The front end reads from the cache to provide answers for client requests. If the cache does not contain the answer, then the front end answers "I don't know" and places the address on the stack. The back end drains the stack by sending questions to a DNS server. The stack is a fixed size, called a "leaky bucket", and will be discussed in more detail in a later section.

The flow of data through the threads and data objects is depicted in Figure 2. A request is read by one of the question threads, which then performs a lookup for the requested IP address in the cache. Any entry found in the cache is used to answer the query: the IP address, its name, and a pointer identifying the i/o stream is placed on an answer queue. If no entry is found in the cache, then a null string is used for the answer and the IP address is placed in the DNS bucket. A fixed number of answer threads drains the answer queue by composing and sending out responses. The DNS question thread drains the DNS bucket, composing queries that ask for PTR records in the domain *in-addr.arpa*. These requests are sent to a name server via UDP. The DNS answer thread reads all DNS replies sent to the process, extracts the domain name and adds an entry to the cache.

A maintenance thread is run at periodic intervals to perform two functions: cache maintenance and persistent storage. Cache maintenance consists of a sweep through the cache to look for entries that have expired. At regular intervals the entire contents of the cache is written to a text file. This provides for a persistent record of the information and allows the cached data to survive server restarts. This file has other uses as well, since it contains address to host mappings of nearly all clients to visit the site in the past several days.

4.3 Protocol

The protocol used between client and server is extremely simple. Although its specification is not significant to the results presented here, a brief description is provided.

The protocol runs over a TCP stream, and an individual connection can handle an unlimited number of requests. A request (from client to server) consists of an IP address represented by 4 octets in network byte order (most significant byte first). Each IP address is separated from the next in the data stream with a framing

octet consisting of all 1's. Should the server get out of sync with the client, it will be able to resync within a few requests.

A response (from server to client) consists of an IP address followed by a null-terminated string. The IP address is formatted as in the request: 4 octets in network byte order. The string is the domain name associated with the IP address and ends with a zero octet. Each response is separated from the next with a framing octet consisting of all 1's. Responses are not coupled with requests: any number of requests can be sent between responses from the server.

There are two peculiarities in this protocol. First, the only identifying information in the response is the IP address itself. This is considered sufficient to match responses with requests, even though it is not unique per request. Second, there is no explicit length given for the variable length response. The client is expected to read until seeing the null octet, and the framing octet is used to ensure that client and server do not get out of sync.

5 Cache Management

The main cache holds answers received from DNS servers. As queries arrive from clients, the answers are served directly out of the cache. As more information is retained in the cache, the likelihood of a cache hit for a given request will increase. But the size of the cache cannot grow without bound due to system memory constraints. A variety of cache management techniques can be used to provide a trade-off between information retention and memory utilization. When we first began this project, we anticipated very high memory utilization on the part of the cache and sized our server system accordingly.

In the current implementation of Rapid DNS, the cache is implemented as a bucket hash keyed on IP address. The number of buckets is fixed throughout the lifetime of the server process, but can be configured at start-up time. The current configuration at CNN uses 400,009 buckets. Each bucket contains a linked list of items which is hashed to the bucket, and there is no limit on the length of each list. Since multiple threads of execution need to access the cache simultaneously, mutex locks are necessary to preserve the integrity of the data structure. Figure 2 clearly shows that only one thread adds data to the cache while multiple threads may be reading information from the cache. To optimize performance, read/write locks[8] were utilized within the cache with one lock being assigned to each bucket in the hash table. Any number of question threads can be reading information from the cache simultaneously. When the DNS answer thread needs to add an entry to the cache, it must first determine the target bucket in the hash table, then it must obtain a write lock on the bucket before inserting the new datum on to the bucket's linked list. A write lock on the bucket must also be obtained before any datum in the bucket is altered or removed.

Although the number of buckets for the cache is constant, the cache is not of a fixed size and a cache management algorithm must be employed to prevent unbounded growth. The current implementation utilizes a first-in first-out (FIFO) algorithm bounded by time. When an entry is added to the cache, it is stamped with an expiration time x seconds in to the future. The configurable value x is the "time to live" and is typically set between three and seven days. Lower settings for time to live result in a smaller run-time cache size. At periodic intervals, a maintenance thread is run that sweeps through the cache and removes any entry beyond its expiration time. The memory used by those entries is returned to the free pool so that it can be used for new entries. In the current implementation the configured time to live is the only value consulted when calculating the expiration time of an entry. The time to live value contained in the response from the DNS server is ignored. Although this policy may degrade the accuracy of the answer, it does reduce the server's dependency on data not under our direct control. If the server used DNS time to live values, then remote servers would have a direct influence on the cache hit ratio. In this application, we prefer an answer that is potentially a few days out of date over no answer at all. Although this policy may have an affect on the statistical results presented below, it certainly has no influence on the implementation. The server could easily be changed to take the minimum of the configured time to live and the DNS time to live when creating the cache entry.

The choice of a strict FIFO cache policy is primarily due to performance concerns. A policy that is tied to the entry creation time does not require modification of the entry at any point during its lifetime. Any policy based on use would require such usage to be tracked, and that question threads modify the entries they were reading to stamp them with a last use time. Such an implementation would require that these threads obtain a write lock before modifying the entry. With a FIFO policy, only the DNS answer thread and the maintenance thread need to obtain write locks. If all the question threads performed write locks, the lock contention would have a detrimental impact on performance.

6 DNS Request Bucket

When a cache miss occurs, the question thread inserts the missing IP address in to a stack. The back end DNS question thread pops addresses off this stack and composes DNS queries for the corresponding PTR records. This data structure is not implemented as a FIFO queue, but as a LIFO stack of bounded size. If a new request fills the stack, then requests at the bottom of the stack are dropped. This "leaky bucket" method prevents an ever increasing backlog of requests while avoiding swamped DNS servers.

Consecutive DNS requests are spaced with a configurable delay to avoid flooding the DNS server. The LIFO implementation intentionally gives priority to most recently received requests. It is expected that web page requests will have a high locality of reference but only for a brief period of time. This is due to two observations. First, the typical web page "view" consists of many embedded images, each of which will generate a separate HTTP request from the same client. Second, it is expected (hoped) that upon seeing the first page the user will be drawn in to the site and request additional pages. So we expect that the longer it has been since we've heard from a client the less likely it is that we will hear from it again in the near future.

The fixed size insures that the stack will not grow without bound. As newer requests enter the bucket, older ones are forced to wait. With each new request the likelihood that the request at the bottom of the bucket will get serviced decreases. Since requests at the bottom of the stack are least likely to ever get serviced they might as well be discarded. The impact of discarding a request in the bucket is that a future request may miss in the cache lookup rather than hit. The result is an "I don't know" answer, which is not a serious concern. As a consequence, however, the IP address will once again be entered at the top of the stack and will have another chance at getting serviced.

A request that misses in the cache will cause an entry to be placed in the bucket, but it might be some time before that request is filled by DNS. During that period of time many more requests for the same address may arrive. To help suppress duplicates in the bucket, the back end will create an empty entry in the cache for the address, giving it two minutes to live. When the back end receives the DNS answer, it simply replaces the null cache entry with the actual data and adjusts the TTL. If the back end sees a request in the bucket for an address that is already in the cache (even if the cache entry is empty), it will not send out a DNS query and instead log the event as a "back end cache hit". Collectively, this technique suppresses the creation of duplicate DNS queries.

7 Negative Caching

A good percentage of the responses from DNS indicate that the requested reverse domain does not exist (response code NXDOMAIN). Consider the consequences of this result on the Rapid DNS server if this result is ignored. After two minutes the blank cache entry which was inserted for duplicate suppression (see Section 6) will expire and additional requests for the address will generate another DNS query. Therefore we considered it prudent to cache this negative information. A response of "no such domain" is represented in the cache with a null entry, and its time to live is set to be the same as regular entries. Further requests for that entry will generate an "I don't know" answer. In this particular case, however, we actually do know that there is no name for this number. From the perspective of the client, the distinction between "not known" and "nonexistent" is unimportant, as they would be logged the same.

During production operation we have noted that a good percentage of the DNS requests generate a failure (response code SERVFAIL). In a typical week we measured that an average 12% of the DNS queries resulted in a SERVFAIL, with a peak at 19.4%. Currently we ignore such responses, and consequently the address that failed will be re-queried in as little as two minutes. The high percentage of such answers implies a potential performance benefit from caching them. Since this is considered to be a transient error, such entries would have to be cached with a much lower time to live than negative answers.

8 Performance Results

Each Rapid DNS server in the CNN web farm routinely handles 250 client connections. Many web servers in the CNN farm are configured to create multiple Netscape processes (or "instances"), and each process needs to open a separate connection. So although there are over 250 client connections on a Rapid DNS server, in reality it may only be serving 50 to 70 machines. The current installation utilizes three Rapid DNS servers, and the clients load balance across the servers. Each server is a Sun SPARC Ultra 2200 with two 200 MHz processors and 1 gigabyte of physical memory. Clients are configured to perform Rapid DNS queries only for http pages (image retrieval does not generate a query).

The cache carries approximately 2.5 million entries (negative information excluded) while the correspond-

ing executable consumes 260 megabytes of virtual memory. Due to the generous server configuration and Unix paging policies, all of the virtual memory typically remains resident. Round trip times for client requests average 2 milliseconds even during heavily loaded periods of the day. Measurements have shown request rates as high as 1,611,180 requests per hour. This corresponds to an average of 26,853 requests per minute or 447 per second.*

8.1 One Week with Standard Configuration

The standard configuration for Rapid DNS servers in the CNN Web Farm uses a hash table with 400,009 slots, a request bucket size of 4096, a 7-day time to live for cache entries, and a DNS query interval of 50 milliseconds. Negative caching is enabled in the standard configuration

Over the course of a one week measuring period with measurements taken hourly, the Rapid DNS server served an average of 5902 requests per minute (98 per second). The peak hourly rate was 862,500 requests, equivalent to 14,375 requests per minute (240 requests per second). During the same measuring period the cache hit ratio ranged from 86.5% to 94.5% with an average ratio of 92.4% and a standard deviation of 1.5%. The measuring period also saw no bucket leaks, except for those caused by a server restart in the middle of the week.

Hourly performance measurements are presented in Figure 3, where the top line shows requests and the lightly shaded area represents cache hits. The diurnal access behavior observed by Gribble and Brewer [5] is evident in these measurements: peak access times for the web servers are reflected in the quantity of Rapid DNS requests.

8.2 Varying the Query Interval

The rate at which the server sends out DNS queries, the query interval, can be tuned to avoid swamping the DNS server during peak loads. Initially this interval was set to 80 milliseconds, but it was discovered that this setting would cause bucket leaks during normal daytime loading.

The query interval defines the rate at which requests in the bucket are processed. If it is too slow the server will be forced to let requests leak out of the bucket. Smaller intervals may send too many requests to the DNS server with the result that some will never get answered. Given a query interval, one can easily compute an upper bound on the frequency of DNS queries that can be accommodated without the risk of leaks. This rate is simply the reciprocal of the frequency:

$$r(f) = \frac{1}{f}$$

Measurements show the expected behavior: a 90 millisecond delay results in a maximum rate of 11.11 requests per second. Given a cache hit ratio of 90% we would expect to see this rate when the number of client requests is approximately 111, a figure that is routinely exceeded during the day. As predicted, a configuration using 90 milliseconds saw a substantial number of bucket leaks. Figure 4 shows an average day running with a 90 ms interval. The top line depicts the number of requests, the lightly shaded area shows the cache hits, and the dark shading indicates the resulting number of bucket leaks. Figure 5 compares the number of actual DNS requests against the leaks for the same period of time. The ceiling on the request rate is evident, and the corresponding leaks show the expected behavior. Subsequent measurements using query intervals between 60 and 80 ms showed negligible leaks during an average week with an acceptable load on the DNS server.

8.3 Effects of Negative caching

To show the benefits of negative caching, we ran one server with negative caching disabled for a one-day period. Figure 6 shows a comparison between this server (the subject) and one which was still caching negative information (the control) for the same time period. Both servers were configured to use a query interval of 50 milliseconds.

It is immediately noticeable that the cache hit rate is significantly lower without negative caching. The subject machine had an average rate of 78.9% while the control saw 90.45%. The subject never exceeded a cache hit rate of 89% and saw a low of 61.2%, while the control had a much more compact range from 84.4% to 93.1%. A comparison of the cache hit rates is given in Figure 7. Because of the lower cache hit rate, more DNS queries enter the bucket and the configuration cannot get all of them out. Consequently there is a corresponding increase in the number of bucket leaks. The subject server peaked at 175,906 leaks in an hour where the control server experienced 75,309 in its worst hour.

Also of interest is the fact that the peak in number of requests for the subject is noticeably lower. This can be attributed to the load balancing code used in the client,

* During early development we ran with only two Rapid DNS servers, and it was common to see hourly rates between 1.5 and 1.6 million.

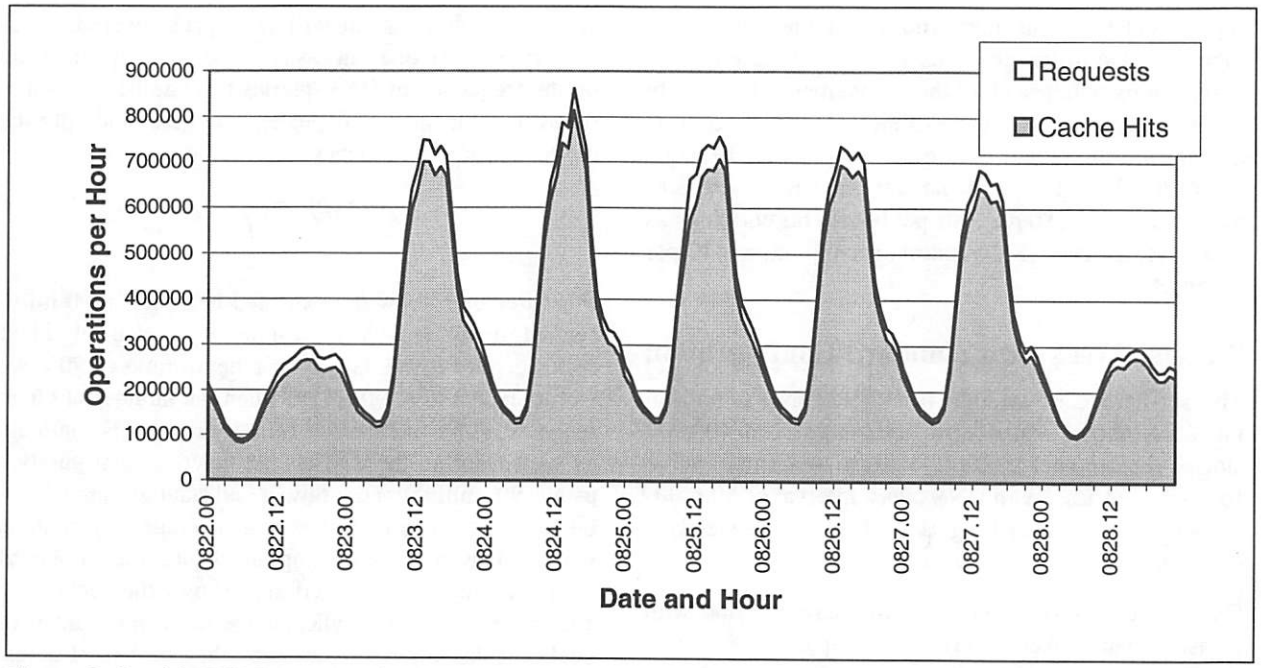


Figure 3: Rapid DNS Server Performance for One Week

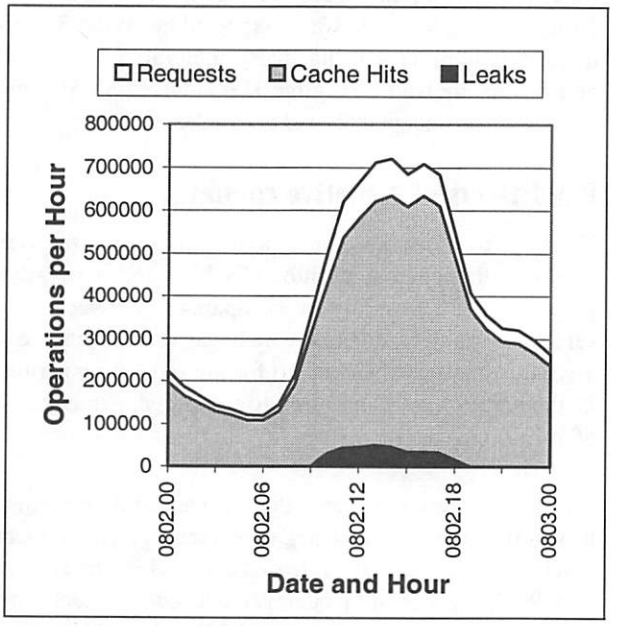


Figure 4: Performance with 90 ms Query Interval

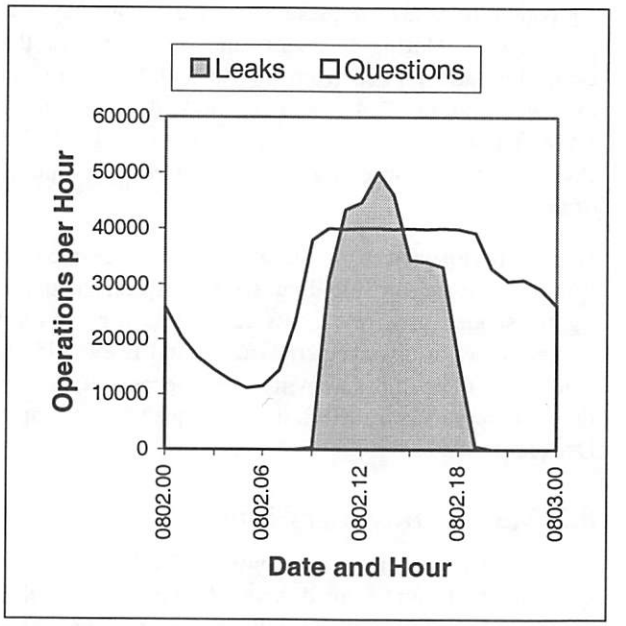


Figure 5: DNS Questions with 90 ms Query Interval

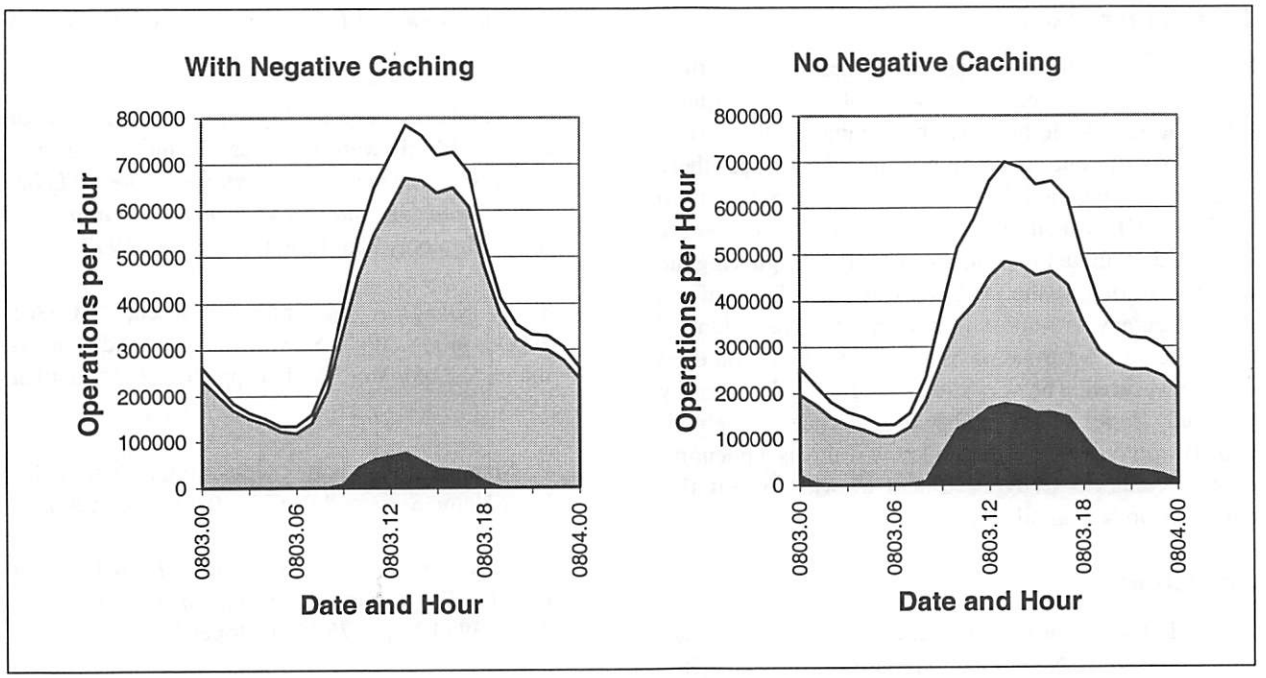


Figure 6: Impact of Negative Caching on Server Performance

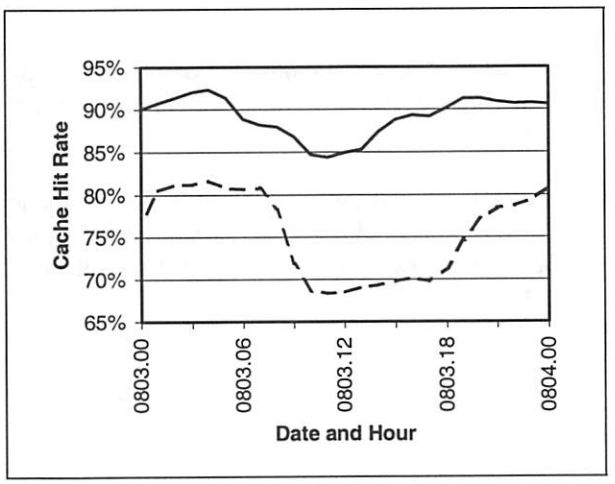


Figure 7: Negative Caching Effects on Cache Hits

which would imply that the average round trip time for the server without negative caching was higher. A higher round trip time would lead to a larger backlog of requests in the client, which in turn would schedule around the backlog and allocate fewer requests to the slower server. The implication is that the added workload caused by processing additional DNS queries and responses had an impact on its ability to provide timely answers to queries.

9 Conclusions

Rapid DNS has provided the CNN Web Farm with a viable mechanism for obtaining client domain names. This ability has enabled it to provide more accurate demographic information and to enable targeting of advertisements by domain name. It is unlikely that direct use of the DNS name servers would have been as effective, given the latency inherent in any domain lookup. The system has met and exceeded expectations since its deployment in March of 1999. One or two web server outages were attributed to the Rapid DNS service, but were caused by simple programming bugs and not by flaws in the design. The system has proven its viability over the course of six months.

Some observations have come out of analysis of the server data.

- The use of negative caching has a marked impact on performance of this particular application.
- Very high cache hit rates have been realized by the system, minimizing direct load on the DNS name servers.
- Even with over 250 client connections, the servers are able to sustain in excess of 400 operations per second.
- A built-in mechanism for throttling outgoing DNS queries controls the load on the name server with a negligible loss of data, especially using query intervals of 80 ms and lower.

10 Further Work

There are several areas that can benefit from further study. The justification for using a FIFO bucket to queue DNS queries should be tested by comparing the performance of different queueing policies. We suspect there would be a measurable improvement in the cache it ratio if SERVFAIL responses (Section 7) were cached, and a study to determine this would be beneficial. Ignoring the time to live field in the DNS reply (Section 5) sacrifices some accuracy to improve performance. The extent of the inaccurate information is not known, but could be easily measured. The very high cache hit ratios seen by this study imply a high locality of reference for web clients. It would be interesting to know if this is a phenomenon peculiar to CNN or if and to what extent this pattern is present at other sites.

Availability

The code for this project was developed under contract with Cable News Network and remains proprietary. It is not available for public distribution.

Acknowledgements

The authors would like to thank Steve Brunton for doing much of the dirty work in installing, baby sitting, and troubleshooting the servers. He performed every software and configuration change we requested without a word of complaint. Thanks are also extended to Paul Holbrook for inspiring us to submit the work. Many of the ideas that went in to Rapid DNS, especially the leaky bucket, are due to Monty Mullig and Sam Gassel.

Bibliography

[1] D. Andresen, T. Yang, V. Holmedahl, O. H. Ibarra, "SWEB: Toward a Scalable World Wide Web Server on Multicomputers," *Proceedings of the 10th International Symposium on Parallel Processing*, pp. 850-856, April 1996.

[2] T. Brisco, "DNS Support for Load Balancing," RFC 1794, April 1995.

[3] V. Cardellini, M. Colajanni, P. Yu, "Redirection Algorithms for Load Sharing in Distributed Webserver Systems," Proceedings of the 19th IEEE International Conference on Distributed Computing Systems, 1999.

[4] P. Danzig, K. Obracza, A. Kumar, "An Analysis of Wide-Area Name Server Traffic," *SIGCOMM '92 Conference Proceedings: Communications, Architectures and Protocols*, pp. 281-292, August 1992.

[5] S. Gribble, E. Brewer, "System Design Issues for Internet Middleware Services: Deductions from a Large Client Trace," *Proceedings of the USENIX Symposium on Internet Technologies and Systems,* Monterey, California, December 1997.

[6] T. T. Kwan, R. McGrath, D. Reed, "NCSA's World Wide Web Server: Design and Performance," *Computer*, **28**(11), pp. 68-74, November 1995.

[7] P. Manning, P. Vixie, "Operational Criteria for Root Name Servers," RFC 2010, October 1996.

[8] P. McKenney, "Selecting Locking Primatives for Parallel Programming," *Communications of the ACM*, **39** (10), pp. 75-82, October 1996.

[9] P. Mockapetris, "Domain Names – Implementation and Specification," RFC 1035, November 1987.

[10] Netscape Communications Corporation, *Netscape Enterprise Server Programmers Guide for Unix*, 1996.

[11] R. Schemers, "lbnamed: A Load Balancing Name Server in Perl," *Proceedings of the Ninth Systems Administration Conference*, pp. 1-11, September 1995.

Connection Scheduling in Web Servers

Mark E. Crovella* Robert Frangioso*
Department of Computer Science
Boston University
Boston, MA 02215
{crovella,rfrangio}@bu.edu

Mor Harchol-Balter†
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3891
harchol@cs.cmu.edu

Abstract

Under high loads, a Web server may be servicing many hundreds of connections concurrently. In traditional Web servers, the question of the order in which concurrent connections are serviced has been left to the operating system. In this paper we ask whether servers might provide better service by using non-traditional service ordering. In particular, for the case when a Web server is serving static files, we examine the costs and benefits of a policy that gives preferential service to short connections. We start by assessing the scheduling behavior of a commonly used server (Apache running on Linux) with respect to connection size and show that it does not appear to provide preferential service to short connections. We then examine the potential performance improvements of a policy that does favor short connections (shortest-connection-first). We show that mean response time can be improved by factors of four or five under shortest-connection-first, as compared to an (Apache-like) size-independent policy. Finally we assess the costs of shortest-connection-first scheduling in terms of unfairness (*i.e.*, the degree to which long connections suffer). We show that under shortest-connection-first scheduling, long connections pay very little penalty. This surprising result can be understood as a consequence of heavy-tailed Web server workloads, in which most connections are small, but most server load is due to the few large connections. We support this explanation using analysis.

1 Introduction

As the demand placed on a Web server grows, the number of concurrent connections it must handle increases. It is not uncommon for a Web server under high loads to be servicing many hundreds of connections at any point in time. This situation raises the question: when multiple outstanding connections require service, in what order should service be provided?

By service, we mean the use of some system device (processor, disk subsystem, or network interface) that allows the server to make progress in delivering bytes to the client associated with the connection. Thus, the question of service order applies collectively to the order in which connections are allowed to use the CPU, the disk(s), and the network interface.

In most Web servers, the question of the order in which concurrent connections should be serviced has typically been left to a general-purpose operating system. The OS scheduler orders access to the CPU, and the disk and network subsystems order service requests for disk and network I/O, respectively. The policies used in these systems typically emphasize fairness (as provided by, *e.g.*, approximately-FIFO service of I/O requests) and favorable treatment of interactive jobs (as provided by feedback-based CPU scheduling).

In this paper, we examine whether Web servers might provide better service by using non-traditional service ordering for connections. In particular, we are concerned with Web servers that serve static files. In this case, the service demand of the connection can be accurately estimated at the outset (*i.e.*, once the HTTP GET has been received) since the size of the file to be transferred is then known; we call this the "size" of the connection. The question then becomes: can servers use the knowledge of connection size to improve mean response time?

Traditional scheduling theory for simple, single-device systems shows that if task sizes are known, policies that favor short tasks provide better mean response time than policies that do not make use of task size. In a single-device system, if run-

*Supported in part by NSF Grant CCR-9706685.
†Supported by the NSF Postdoctoral Fellowship in the Mathematical Sciences.

ning jobs can be pre-empted, then the optimal work-conserving policy with respect to mean response time is *shortest remaining processing time first* (SRPT). Since a Web server is not a single-device system, we cannot use SRPT directly. However we can employ service ordering within the system that attempts to approximate the effects of SRPT.

The price to be paid for reducing mean response time is that we reduce the fairness of the system. When short connections are given favorable treatment, long connections will suffer. Care must be taken to ensure that the resulting unfairness does not outweigh the performance gains obtained.

Our goal in this paper is to explore the costs and benefits of service policies that favor short connections in a Web server. We call this the *connection scheduling* problem. The questions we address are:

1. How does a traditional Web server (Apache running on Linux) treat connections with respect to their size? Does it favor short connections?

2. What are the potential performance improvements of favoring short connections in a Web server, as compared to the traditional service order?

3. Does favoring short connections in a web server lead to unacceptable unfairness?

We are interested in the answers to these questions in the context of a traditionally structured operating system (like Linux). Thus, to answer these questions we have implemented a Web server, running on Linux, that allows us to experiment with connection scheduling policies. For each of the devices in the system (CPU, disk, and network interface) the server allows us to influence the order in which connections are serviced. Since all of our scheduling is done at the application level, our server does not allow us precise control of all of the components of connection service, particularly those that occur in kernel mode; this is a general drawback of traditional operating systems structure whose implications we discuss in detail below. However our server does provide sufficient control to allow us to explore two general policies: 1) *size-independent* scheduling, in which each device services I/O requests in roughly the same order in which they arrive; and 2) *shortest-connection-first* scheduling, in which each device provides service only to the shortest connections at any point in time. We use the SURGE workload generator [5] to create Web requests; for our purposes, the important property of SURGE is that it accurately mimics the size distribution of requests frequently seen by Web servers.

Using this apparatus, we develop answers to the three questions above. The answer to our first question is that Apache does *not* appear to favor short connections. We show that compared to our server's size-independent policy, Apache's treatment of different connection sizes is approximately the same (and even can be more favorable to long connections—a trend opposite to that of shortest-connection-first).

This result motivates our second question: How much performance improvement is possible under a shortest-connection-first scheduling policy, as compared to size-independent (*i.e.,* Apache-like) scheduling? We show that for our server, adopting shortest-connection-first can improve mean response time by a factor of 4 to 5 under moderate loads.

Finally our most surprising result is that shortest-connection-first scheduling does *not* significantly penalize long connections. In fact, even very long connections can experience improved response times under shortest-connection-first scheduling when compared to size-independent scheduling. To explore and explain this somewhat counterintuitive result we turn to analysis. We use known analytic results for the behavior of simple queues under SRPT scheduling, and compare these to size-independent scheduling. We show that the explanation for the mild impact of SRPT scheduling on long connections lies in the size distribution of Web files—in particular, the fact that Web file sizes show a *heavy tailed* distribution (one whose tail declines like a power-law). This result means that Web workloads are particularly well-suited to shortest-connection-first scheduling.

2 Background and Related Work

The work reported in this paper touches on a number of related areas in server design, and in the theory and practice of scheduling in operating systems.

Traditional operating system schedulers use heuristic policies to improve the performance of short tasks given that task sizes are not known in advance. However, it is well understood that in the case where the task sizes *are* known, the work-conserving scheduling strategy that minimizes mean response time is shortest-remaining-processing-time first (SRPT). In addition to SRPT, there are many algorithms in the literature which are designed for the case where the task size is known. Good overviews of the single-node scheduling problem and

its solution are given in [7], [14], and [17].

In our work we focus on servers that serve static content, *i.e.,* files whose size can be determined in advance. Web servers can serve dynamic content as well; in this case our methods are less directly applicable. However, recent measurements have suggested that most servers serve mainly static content, and that dynamic content is served mainly from a relatively small fraction of the servers in the Web [15].

Despite the fact that the file sizes are typically available to the Web server, very little work has considered size-based scheduling in the Web. One paper that does discuss size-based scheduling in the Web is that of Bender, Chakrabarti, and Muthukrishnan [6]. This paper raises an important point: in choosing a scheduling policy it is important to consider not only the scheduling policy's performance, but also whether the policy is fair, *i.e.* whether some tasks have particularly high slowdowns (where slowdown is response time over service time). That paper considers the metric *max slowdown* (the maximum slowdown over all tasks) as a measure of unfairness. The paper proposes a new algorithm, *Dynamic Earliest Deadline First (DEDF)*, designed to perform well on both the mean slowdown and max slowdown metrics. The DEDF algorithm is a theoretical algorithm which cannot be run within any reasonable amount of time (it requires looking at all previous arrivals), however it has significance as the first algorithm designed to simultaneously minimize max slowdown and mean slowdown. That work does consider a few heuristics based on DEDF that are implementable; however, simulation results evaluating those more practical algorithms at high load indicate their performance to be about the same as SRPT with respect to max slowdown and significantly worse than SRPT with respect to mean slowdown.

At the end of our paper (Section 6) we turn to analysis for insight into the behavior of the shortest-connection-first scheduling policy. In that section we examine a single queue under SRPT scheduling, which was analyzed by Schrage and Miller [18].

In addition to scheduling theory, our work also touches on issues of OS architecture. In particular, the work we describe in this paper helps to expose deficiencies in traditional operating system structure that prevent precise implementation of service policies like shortest-connection-first. Shortest-connection-first scheduling requires that resource allocation decisions be based on the connection requiring service. This presents two problems: first, kernel-space resource allocation is not under the control of the application; and second, resource allocation is difficult to perform on a per-connection basis. These two problems have been have been noted as well in other work [1, 3].

3 A Server Architecture for Scheduling Experiments

In this section we present the architecture of the web server we designed and developed for use in our experiments. Our primary goal in designing this server was to provide the ability to study policies for scheduling system resources. Two additional but less important goals were simplicity of design and high performance.

3.1 Scheduling Mechanisms

Web servers like Apache [11] follow the traditional architectural model for Internet service daemons, in which separate connections are served by separate Unix processes. This model is insufficient for our needs because none of its scheduling decisions are under application control. Instead we need to expose scheduling decisions to the application as much as possible.

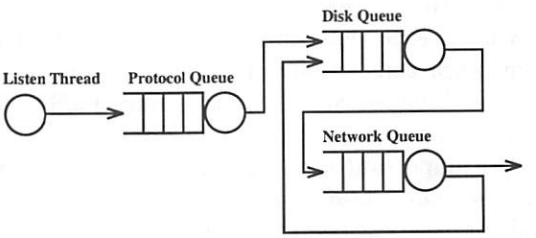


Figure 1: Organization of the Experimental Server

A typical connection needs three types of service after connection acceptance: 1) protocol processing, 2) disk service, and 3) network service. In order to expose the scheduling decisions associated with each type of service, we organize the server application as a set of three queues. This architecture is shown in Figure 1. The entities that are held in (and move between) queues correspond to individual connections. Next we describe how, by varying the service order of each queue, the application can influence resource scheduling decisions on a per-connection basis.

Each queue in Figure 1 has an associated pool of threads. In addition there is a single listen thread. The role of the listen thread is to block on the `accept()` call, waiting for new connections. When a new connection arrives, it creates a *connection descriptor*. The connection descriptor encapsulates the necessary state the connection will need (two file

descriptors, a memory buffer, and progress indicators). It then places the connection descriptor into the protocol queue and resumes listening for new connections.

Protocol threads handle all aspects of HTTP. When a protocol thread is done, the server knows what file is being requested and is ready to start sending data. In addition the thread has called `stat()` on the file to determine the file's size. The protocol thread then enqueues the connection descriptor into the disk queue.

The role of the disk thread is to dequeue a connection descriptor, and based on the file associated with the connection, `read()` a block of file data from the filesystem into the connection descriptor's buffer. Currently our server reads blocks of up to 32KB at a time. The `read()` call is blocking; when it returns, the thread enqueues the descriptor into the network queue.

The network thread also starts by dequeueing a connection descriptor; it then calls `write()` on the associated socket to transfer the contents of the connection's buffer to the kernel's socket buffer. The `write()` call is blocking. When it returns, if the all the bytes in the file have been transferred, the network thread will close the connection; otherwise it will place the descriptor back into the disk queue. Thus each connection will move between the network and disk queues until the connection has been entirely serviced.

An important advantage of this architecture is that we can observe which subsystem (protocol, disk, or network) is the bottleneck by inspecting the lengths of the associated queues. For the workloads we used (described in Section 4.1) we found that the bottleneck was the network queue; queue lengths at the protocol and disk queues were always close to zero.

This scheme gives us a flexible environment to influence connection scheduling by varying the service order at each queue. In this study we focus on two scheduling policies: *size-independent* and *shortest-connection-first.* In our size-independent policy, each thread simply dequeues items from its associated queue in FIFO order. The implication of this policy is that each connection is given a fair share of `read()` and `write()` calls, and that any fixed set of connections is conceptually served in approximately round-robin order.

Under shortest-connection-first scheduling, each (disk or network) thread dequeues the connection that has the least number of bytes remaining to be served. This appears to be a good indicator of the remaining amount of work needed to service the connection. More precise policies are possible, which are not considered in this paper.

Finally, we note that the listen and protocol threads run at a higher priority than disk and network threads, and that the protocol queue is always served in FIFO order (since connection size is not yet known).

3.2 Performance

As stated at the outset, high performance is only a secondary goal of our archtectural design. However our architecture is consistent with recent trends in high performance servers and (as shown in Section 5) yields performance that is competitive with a more sophisticated server (Apache).

Considerable attention has been directed to improving the architecture of high performance web servers. As mentioned above, servers like Apache follow the model in which separate connections are served by separate Unix processes. More recent servers have moved away from process-per-connection model, toward lower-overhead strategies. A number of Web server architectures based on a single or fixed set of processes have been built [13, 16]. Removing the overhead of process creation, context switching, and inter-process-communication to synchronize and dispatch work allows the server to use system resources more efficiently. In addition, in single-process servers memory consumption is reduced by not using a running process for each concurrent connection receiving service, which allows such servers to make more effective use of memory to cache data. The drawback is that single process web servers are typically more complicated and must rely on multi-threading or non-blocking I/O schemes to achieve high throughput.

Our server obtains the performance benefits of using a single process, with a fixed number of threads (*i.e.,* it does not use a thread per connection). However we have not adopted all of the performance enhancements of aggressively optimized servers like Flash [16] because of our desire to keep the server simple for flexibility in experimenting with scheduling policies. In particular we use blocking threads for writing, which is not strictly necessary given an nonblocking I/O interface. However we note that the policies we explore in our server appear to be easily implementable in servers like Flash.

3.3 Limitations

The principal limitation of our approach is that we do not have control over the order of events inside the operating system. As a specific exam-

ple, consider the operation of the network subsystem. Our server makes `write()` calls which populate socket buffers in the network subsytem in a particular order. However, these buffers are not necessarily drained (*i.e.,* written to the network) in the order in which our application has filled them.

Two factors prevent precise control over the order in which data is written to the network. First, each buffer is part of a flow-controlled TCP connection with a client. If the client is slow with respect to the server, the client's behavior can influence the order in which buffers are drained. For this reason in our experiments we use a multiple high-performance clients and we ensure that the clients are not heavily loaded on average. Thus in our case client interaction is not a significant impediment to scheduling precision.

The second, more critical problem is that in traditional Unix network stack implementations, processing for all connections is handled in an aggregate manner. That is, outgoing packets are placed on the wire in response to the arrival of acknowledgements. This means that if many connections have data ready to send, and if the client and network are not the bottleneck, then data will be sent from the set of connections in order that acknowledgements arrive, which is not under application control. The implication is that if the network subsystem has a large number of connections that have data ready to send, then the order in which the application has written to socket buffers will have less effect on the scheduling of connections.

The problem has been recognized and addressed in previous work. In particular, Lazy Receiver Processing [10] can isolate each connection's path through the network stack. This allows scheduling decisions to be made on a per-connection basis at the level of the network interface.

Our goal was to demonstrate the improvements possible without operating system modification. As a result, to obtain control over I/O scheduling we limit the concurrency in the I/O systems. For example, by limiting concurrency in the network subsystem, we limit the number of connections that have data ready to send at any point in time, thus narrowing the set of connections that can transmit packets. Because our disk and network threads use blocking I/O, we need multiple threads if we want to have concurrent outstanding I/O requests. This means that it is straightforward to control the amount of concurrency we allow in the kernel subsystems, by varying the number of threads in each of the pools.

At one extreme, if we allow only one thread per pool, then we have fairly strict control over the order of events inside the kernel. At any point in time the kernel can only have one I/O request of each type pending, so there are no scheduling decisions available to it. Unfortunately this approach sacrifices throughput; both the disk and network subsystems make use of concurrent requests to overlap processing with I/O. At the other extreme, if we provide a large number of threads to each pool, we can obtain high throughput; however then we lose all control over scheduling.

In order to explore the utility of shortest-connection-first scheduling, we have adopted an intermediate approach. Rather than running our server at its absolute maximum throughput (as measured in bytes per unit time), we limit its throughput somewhat in order to obtain control over I/O scheduling. Note however that our server's throughput (in bytes per second) is still greater than that of Apache under the same load level.[2] The performance implications of this approach are presented in Section 5.4. It is important to note that this is only necessary because of the limitations of the traditionally structured OS on which we run our experiments, and this restriction could be dropped given a different OS structure. Our approach thus allows us enough influence over kernel scheduling to demonstrate the costs and benefits of the shortest-connection-first policy.

4 Experimental Setup

4.1 File Size Distribution

An important aspect of our work is that we have focused on careful modeling of the file size distribution typically seen on Web servers. As shown in Section 6, properties of the file size distribution are directly related to some of our results.

Our starting point is the observation that file sizes on Web servers typically follow a *heavy-tailed* distribution. This property is surprisingly ubiquitous in the Web; it has been noted in the sizes of files requested by clients, the lengths of network connections, and files stored on servers [2, 8, 9]. By heavy tails we mean that the tail of the empirical distribution function declines like a power law with exponent less than 2. That is, if a random variable X follows a heavy-tailed distribution, then

$$P[X > x] \sim x^{-\alpha}, \quad 0 < \alpha < 2$$

where $f(x) \sim a(x)$ means that $\lim_{x\to\infty} f(x)/a(x) = c$ for some positive constant c.

[2] Apache was configured for high performance as described in Section 4.2.

	Body
Distribution	Lognormal
PMF	$\frac{1}{x\sigma\sqrt{2\pi}}e^{-(\ln x-\mu)^2/2\sigma^2}$
Range	$0 \leq x < 9020$
Parameters	$\mu = 7.630$; $\sigma = 1.001$

	Tail
Distribution	Bounded Pareto
PMF	$\frac{\alpha k^\alpha}{1-(k/p)^\alpha}x^{-\alpha-1}$
Range	$9020 \leq x \leq 10^{10}$
Parameters	$k = 631.44$; $\alpha = 1.0$; $p = 10^{10}$

Table 1: Empirical Task Size Model. PMF is the probability mass function, $f(x)$, where $\int_a^b f(x)dx$ represents the probability that the random variable takes on values between a and b.

Random variables that follow heavy tailed distributions typically show extremely high variability in size. This is exhibited as many small observations mixed with a small number of very large observations. The implication for Web files is that a tiny number of the very largest files make up most of the load on a Web server. We refer to this as the *heavy-tailed property* of Web task sizes; it is central to the discussion in this paper and will come up again in Section 6.

Although Web files typically show heavy tails, the body of the distribution is usually best described using another distribution. Recent work has found that a hybrid distribution, consisting of a body following a lognormal distribution and a tail that declines via a power-law, seems to fit well some Web file size measurements [4, 5]. Our results use such a model for task sizes, which we call the *empirical* model; parameters of the empirical model are shown in Table 1. In this empirical file size model, most files are small—less than 5000 bytes. However, the distribution has a very heavy tail, as determined by the low value of α in the Bounded Pareto distribution, and evidenced by the fact that the mean of this distribution is 11108 — much larger than the typical file size.

4.2 Experimental Environment

To generate HTTP requests that follow the size distribution described above, we use the SURGE workload generator [5]. In addition to HTTP request sizes, SURGE's stream of HTTP requests also adheres to empirically derived models for the sizes of files stored on the server; for the relative popularity of files on the server; for the temporal locality present in the request stream; and for the timing of request arrivals at the server.

SURGE makes requests using synthetic clients, each of which operates in a loop, alternating between requesting a file and lying idle. Each synthetic client is called a *User Equivalent* (UE). The load that SURGE generates is varied by varying the number of UEs. In our tests we varied the number of UEs from 400 to 2000. Validation studies of SURGE are presented in [5]; that paper shows that the request stream created by SURGE conforms closely to measured workloads and is much burstier, and hence more realistic, than that created by SPECWeb96 (a commonly used Web benchmarking tool).

All measurements of Apache's performance presented in this paper were generated using version 1.2.5. We configured Apache for high performance as recommended on Apache's performance tuning Web page.[3] In particular, `MaxRequestsPerChild` was set to 0, meaning that there is no fixed limit to the number of connections that can be served by any of Apache's helper processes. This setting improves Apache's performance considerably as compared to the default.

In addition to the data reported in this paper, we also ran many experiments using version 1.3.4 of Apache. Our experiments indicated that this version had a performance anomaly under low load that we did not isolate, so we do not present those results. However, our experiments indicated that although version 1.3.4 was somewhat faster for short connections, its overall performance was not different enough to affect the conclusions in this paper.

All our tests were conducted using two client machines (evenly splitting the SURGE UEs) and one server in a 100 Mbit switched Ethernet environment. After our experiments were concluded, we found that one client's port was malfunctioning and delivering only 10 Mbit/sec; so half of the load was generated over a path that was effectively only 10Mbit/sec, while the other half of the load arrived over a 100Mbit/sec path. All machines were Dell Dimensions equipped with Pentium II 233 processors, 128 MB of RAM, and SCSI disks. Each of these machines was running Linux 2.0.36. We configured SURGE to use a file set of 2000 distinct files varying in size from 186 bytes to 121 MB. Our measurements pertaining to response time, byte throughput, and HTTP GETs per second were extracted from client side logs generated by SURGE.

All experiments were run for ten minutes. This time was chosen to be sufficiently long to provide

[3] `http://www.apache.org/docs/misc/perf-tuning.html`.

confidence that the measurements were not strongly influenced by transients. For a 1000 UE experiment, this meant that typically more than 2.5GB of data was transferred and more than 250,000 connections took place.

5 Results

5.1 Characterizing Apache's Performance as a Function of Task Size

In this section we characterize Apache's performance as a function of task size under 3 different load levels: 1000 UEs, 1400 UEs, and 1800 UEs. These loads correspond to lightly loaded, moderately loaded, and overloaded conditions for the server. Thus they span the range of important loads to study.

In order to study how different servers treat connections with respect to size, we bin HTTP transactions according to size, and plot the mean response time over all transactions in the bin, as a function of mean file size of transactions in the bin. We plot the resulting data on log-log axes, in order to simultaneously examine both very small and very large connection sizes. Bin sizes grow exponentially, leading to equal spacing on logarithmic axes.

Figure 2 shows the resulting plots of mean response time as a function of file size under Apache and under our Web server with size-independent scheduling for the three different load levels. The plots generally show that reponse time of small files (less than about 10KB) is nearly independent of file size. For this range of files, response time is dominated by connection setup cost. For large files (larger than about 10KB) reponse time increases as a function of file size in an approximately linear manner.

Across all loads, two trends are evident from the figure. First, for small files, Apache tends to provide the same response time or worse response time than does our size-independent server. Second, for large files, Apache tends to provide the same reponse time or better than does our size-independent server.

These two trends indicate that with respect to size, Apache treats connections in a manner that is either approximately the same as our size-independent policy, or else is more favorable to long connections.[4] That is, Apache is, if anything, *punishing* short connections with respect to our size-independent server.

[4] As discussed in Section 4.2 these measurements use Apache version 1.2.5. We found in other experiments with Apache 1.3.4 (not shown here) that the newer version of Apache in fact shows performance that is even closer to that of our server with size-independent scheduling.

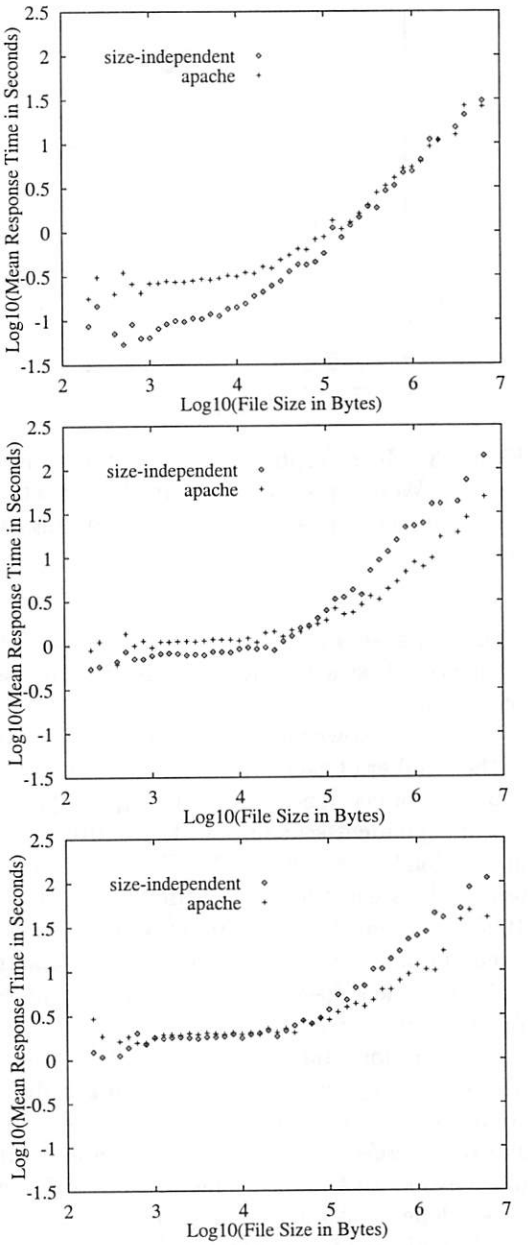


Figure 2: Mean transfer time as a function of file size under the Apache server and our server with size-independent scheduling. Both axes use a log scale. Top to bottom: 1000, 1400, and 1800 UEs.

5.2 Performance Improvements Possible with Shortest-Connection-First Scheduling

Given that Apache does not appear to treat connections in a manner that is favorable to short connections, our next question is whether a policy that does favor short connections leads to performance improvement, and if so, how much improvement is possible. Thus, in this section we compare the

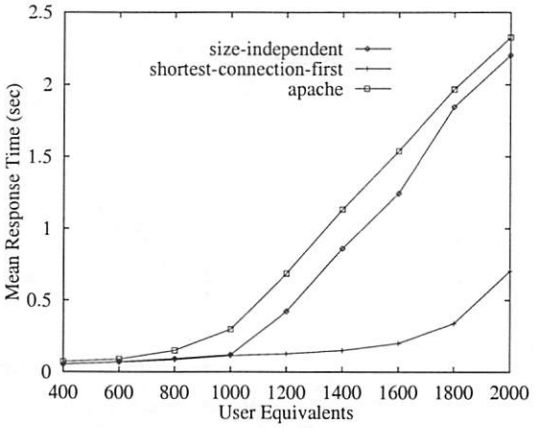


Figure 3: Mean response time as a function of load for our Web server with shortest-connection-first scheduling and size-independent scheduling, and for Apache.

mean response time of our Web server with shortest-connection-first scheduling versus size-independent scheduling.

Figure 3 shows mean response time as a function of the number of user equivalents for our server with shortest-connection-first scheduling compared with size-independent scheduling. The figure shows that at low loads (less than 1000 UEs) there is no difference between the two scheduling policies. Thus 1000 UEs represents the point where the network queue in our server first starts to grow, making the order in which it services write requests important. As the load increases beyond 1000 UEs, the difference in performance between shortest-connection-first scheduling and size-independent scheduling becomes stark. For example under 1400 UEs, the shortest-connection-first scheduling policy improves mean response time by a factor of 4 to 5 over the size-independent scheduling policy.

Also plotted for reference in Figure 3 is the mean response time of Apache under the same conditions. As can be seen, Apache's performance is very similar to that of our server with size-independent scheduling. This is consistent with the conclusions from the previous subsection.

It is important to note that these performance figures may only be lower bounds on the improvement possible by using shortest-connection-first scheduling, due to our constraint of working within a traditionally structured operating system.

5.3 How Much Do Long Connections Suffer?

In the previous section we saw that large improvements in mean transfer time were possible by running our Web server under shortest-connection-first scheduling as opposed to size-independent scheduling. The question now is: does this performance improvement come at a significant cost to large jobs? Specifically, we ask whether large jobs fare worse under shortest-connection-first scheduling than they do under size-independent scheduling.

To answer this question we examine the performance of both shortest-connection-first scheduling and size-independent scheduling as a function of task size. The results are shown in Figure 4, again for a range of system loads (UEs).

The figure shows that in the case of 1000 UEs, shortest-connection-first scheduling is identical in performance to size-independent scheduling across all file sizes. Thus since there is no buildup at the network queue, there is also no performance improvement from shortest-connection-first scheduling in the case of 1000 UEs. However, the figure shows that in the case of 1400 UEs, shortest-connection-first results in much better performance for small jobs (as compared with size-independent scheduling), and yet the large jobs still do not fare worse under shortest-connection-first scheduling than they do under size-independent scheduling. Thus the overall performance improvement does *not* come at a cost in terms of large jobs. When we increase the load to 1800 UEs, however, the large jobs do begin to suffer under shortest-connection-first scheduling as compared with size-independent scheduling. In fact over all our experiments, we find that in the range from about 1200 UEs to 1600 UEs, shortest-connection-first allows short connections to experience considerable improvement in response time without significantly penalizing long connections. This seemingly counter-intuitive result is explained, with analytical justification, in Section 6.

5.4 Varying The Write Concurrency

As discussed in Section 3.3, in order to gain control over the order in which data is sent over the network, we need to restrict our server's throughput (in bytes per second). In this section we quantify this effect.

In all our previous plots we have used a thread pool of 35 threads to service the network queue. Figure 5 shows the effect of varying the number of network threads on mean response time and on documents (HTTP GETs) served per second, at a load of 1400 UEs.

Both plots in the figure make the point that as the number of write threads increases, the difference in performance between shortest-connection-first and size-independent scheduling decreases, un-

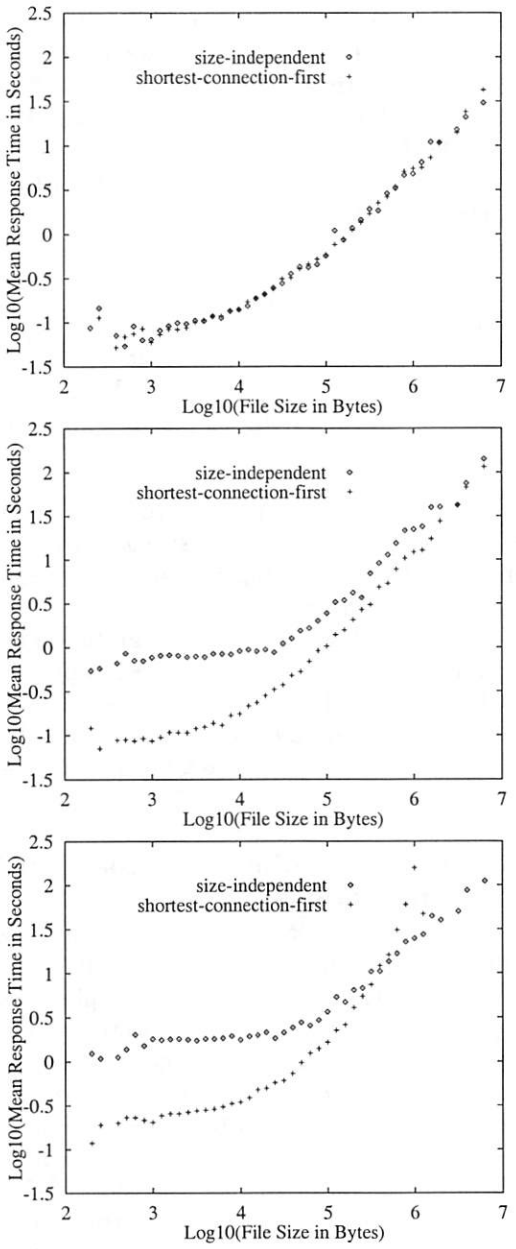


Figure 4: Response time as a function of task size for shortest-connection-first scheduling versus size-independent scheduling. Top to bottom: 1000, 1400, and 1800 UEs.

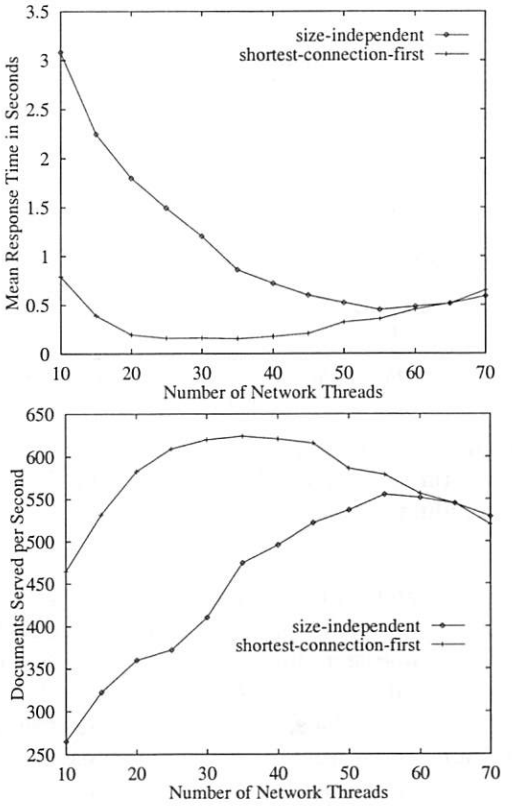


Figure 5: The effect of varying the number of network threads on (upper) mean response time and (lower) HTTP GETs/second for the two scheduling policies.

til at about 60 threads, the choice of scheduling policy has no effect. At the point of 60 threads, there is no buildup in the network queue and all scheduling is determined by kernel-level events.

These plots also show that as the number of threads used declines from 35 threads, the performance difference between shortest-connection-first and traditional scheduling becomes even greater. This suggests that the advantage of shortest-connection-first scheduling may be even more dramatic in a system where there is greater control over kernel-level scheduling, since as the number of threads declines in our system, the degree of control over kernel scheduling increases.

Figure 6 shows the effect on byte throughput of varying the number of network threads. In this figure we have plotted the total number of bytes in files succesfully transferred during each of our 10 minute experiments. It shows that for our server, throughput increases roughly linearly with additional network threads, regardless of the policy used. In all cases, shortest-connection-first has slightly higher throughput than our size-independent policy; this is because the server is serving fewer concurrent connections (on average) and so can provide slightly higher aggregate performance. The primary point to note is when configured with 35 network threads, our server is not performing at peak throughput; this is the price paid for control over network scheduling. As stated earlier, this price is exacted because the kernel does not support per-connection scheduling within the protocol stack.

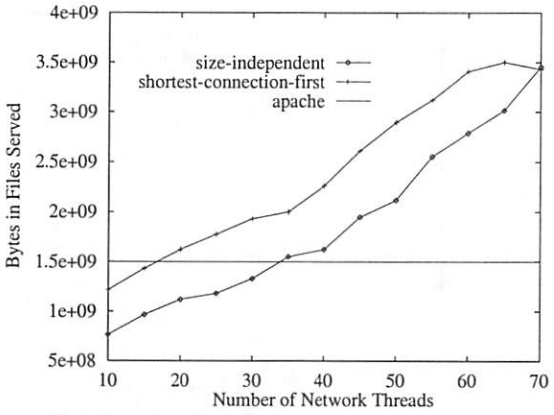


Figure 6: The effect of varying the number of network threads on the byte throughput for the two scheduling policies.

Also plotted for reference in Figure 6 is the corresponding byte throughput of Apache at 1400 UEs. The comparison illustrates that, for 35 network threads, our server is achieving higher throughput than Apache. Thus, although using 35 threads limits our server from its maximum possible performance, it is a level that still outperforms Apache.

6 Analysis: Why Don't Large Jobs Suffer?

In this section we help explain why long connections aren't severely penalized in our experiments using some simple analytic models. These models are only approximations of the complex systems comprising a Web server, but they yield conclusions that are consistent with our experimental results and, more importantly, allow us to explore the reasons behind those experimental results. The results in this section are based on [12]; that paper presents additional background and more results not shown here.

6.1 Assumptions Used in Analysis

In our analysis we examine the $M/G/1$ queue, which is a simple queue fed by a Poisson arrival stream with an arbitrary distribution of service times. The service order is shortest-remaining-processing-time first (SRPT).

Under the SRPT model, only one task at each instant is receiving service, namely, the task with the least processing time remaining. When a new task arrives, if its service demand is less than the remaining demand of the task receiving service, the current task is pre-empted and the new task starts service. We use SRPT as an idealization of shortest-connection-first scheduling because in both cases, tasks with small remaining processing time are always given preference over tasks with longer remaining processing time.

Our analytical results throughout are based on the following equation for the mean response time for a task of size x in an $M/G/1$ queue with load ρ, under SRPT [18]:

$$E\{R_x^{SRPT}\} = \frac{\lambda \int_0^x t^2\, dF(t) + \lambda x^2 (1 - F(x))}{2\left(1 - \lambda \int_0^x t\, dF(t)\right)^2} + \int_0^x \frac{1}{(1 - (\lambda \int_0^t z\, dF(z)))} dt$$

where R_x is the response time (departure time minus arrival time) of a job of size x, $F(\cdot)$ is the cumulative distribution function of service time, and λ is the arrival rate.

We adopt the assumption that the amount of work represented by a Web request is proportional to the size of the file requested. Thus, we use as our task size distribution the empirical file size distribution as shown in Table 1 (the same as that generated by SURGE).

6.2 Understanding the Impact on Long Connections

Using our simple models we can shed light on why large tasks do not experience significant penalties under SRPT scheduling; the explanation will apply equally well to long connections in a Web server employing shortest-connection-first.

The key observation lies in the *heavy-tailed* property of the workload being considered. As defined in Section 4.1, this means that a small fraction of the largest tasks makes up most of the arriving work. The Bounded Pareto distribution, which makes up the tail of our empirical distribution, is extremely heavy-tailed. For example, for a Bounded Pareto distribution with $\alpha = 1.1$, the largest 1% of all tasks account for more than half the total service demand arriving at the server. For comparison, for an exponential distribution with the same mean, the largest 1% of all tasks make up only 5% of the total demand. Now consider the effect on a very large task arriving at a server, say a task in the 99th percentile of the job size distribution. Under the Bounded Pareto distribution, this task in the 99th percentile is interrupted by less than 50% of the total work arriving. In comparison, under the exponential distribution, a task in the 99th percentile of the job size distribution is interrupted by about 95% of the total work arriving. Thus, under the

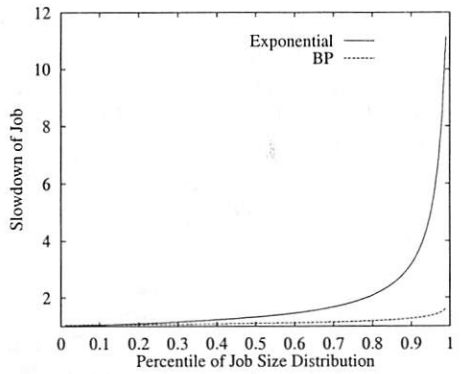


Figure 7: Mean slowdown under SRPT as a function of task size; $\rho = 0.9$.

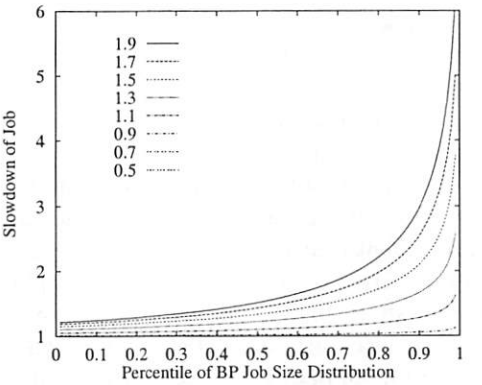


Figure 8: Mean slowdown under SRPT as a function of task size, varying α of task size distribution.

heavy-tailed distribution, a large job suffers *much* less than under a distribution like the exponential. The rest of this section explains and supports this observation.

To evaluate the potential for unfairness to large tasks we plot the mean slowdown of a task of a given size, as a function of the task size. Slowdown is defined as the ratio of a task's response time to its service demand. Task size is plotted in percentiles of the task size distribution, which allows us to assess what fraction of largest tasks will achieve mean slowdown greater than any given value.

Figure 7 shows mean slowdown as a function of task size under the SRPT discipline, for the case of $\rho = 0.9$. The two curves represent the case of an exponential task size distribution, and a Bounded Pareto (BP) task size distribution with $\alpha = 1.1$. The two distributions have the same mean.

Figure 7 shows that under high server load ($\rho = 0.9$), there can be considerable unfairness, but *only for the exponential distribution.* For example, the largest 5% of tasks under the exponential distribution all experience mean slowdowns of 5.6 or more, with a non-negligible fraction of task sizes experiencing mean slowdowns as high as 10 to 11. In contrast, *no* task size in the BP distribution experiences a mean slowdown of greater than 1.6. Thus, when the task size distribution has a light tail (exponential), SRPT can create serious unfairness; however when task size distributions show a heavy tail (BP distribution), SRPT does not lead to significant unfairness.

To illustrate the effect of the heavy-tailed property on the degree of unfairness experienced by large jobs, we plot mean slowdown as a function of task size over a range of BP task size distributions with constant mean (in this case, 3000) and varying α. This plot is shown in Figure 8. The high α cases represent relatively light tails, whereas the low α cases represent relatively heavy tails in the task size distribution.

This figure shows how the degree of unfairness under SRPT increases as the tail weight of the task size distribution decreases. When α is less than about 1.5, there is very little tendency for SRPT to penalize large tasks (curves for $\alpha = 0.5$ and $\alpha = 0.7$ stay so close to 1 as to be invisible on the plot). Only as α gets close to 2.0 (*e.g.*, 1.7 or 1.9) is there any significant fraction of tasks that experience high mean slowdowns.

These figures show that as the heavy-tailed property grows more pronounced, unfairness in the system under SRPT diminishes. Thus the explanation for the surprising resistance of the heavy-tailed task size distributions to unfairness under SRPT is an effect of the heavy-tailed property.

7 Conclusion

This paper has suggested that Web servers serving static files may show significant performance improvements by adopting nontraditional service ordering policies. We have examined the behavior of a popular Web server (Apache running on Linux) and found that, with respect to connection size, it appears to provide service similar to a server that uses size-independent scheduling. Furthermore, we have found that significant improvements in mean response time—on the order of factors of 4 to 5—are achievable by modifying the service order so as to treat short connections preferentially. Finally, we have found that such a service ordering does *not* overly penalize long connections. Using analysis, we have shed light on why this is the case, and concluded that the heavy-tailed properties of Web workloads (*i.e.*, that a small fraction of the longest connections make up a large fraction of the total

work) make Web workloads especially amenable to shortest-connection-first scheduling.

Our work has a number of limitations and directions for future work. The architecture of our server does not allow us precise control over the scheduling of kernel-mode operations (such as I/O). This prevents us from determining the exact amount of improvement that is possible under scheduling policies that favor short connections. We plan to implement short-connection favoring strategies over a kernel architecture that is better designed for server support [3, 10] in order to assess their full potential.

There is room for better algorithmic design here, since the policy we have explored does not prevent the starvation of jobs in the case when the server is *permanently* overloaded. One commonly adopted solution to this problem is dynamic priority adjustment, in which a job's priority increases as it ages, allowing large jobs to eventually obtain priorities equivalent to those of small jobs. We plan to explore such improved policies, perhaps following the initial work in [6].

While more work needs to be done, our results suggest that nontraditional scheduling order may be an attractive strategy for Web servers that primarily serve static files. In particular, the fact that Web workloads (file sizes and connection lengths) typically show heavy-tailed distributions means that shortest-connection-first policies can allow Web servers to significantly lower mean response time without severely penalizing long connections.

References

[1] J. Almeida, M. Dabu, A. Manikutty, and P. Cao. Providing differentiated quality-of-service in Web hosting services. In *Proceedings of the First Workshop on Internet Server Performance*, June 1998.

[2] Martin F. Arlitt and Carey L. Williamson. Internet web servers: Workload characterization and performance implications. *IEEE/ACM Transactions on Networking*, 5(5):631–645, 1997.

[3] Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul. Resource containers: A new facility for resource management in server systems. In *Proceedings of OSDI '99*, pages 45–58, 1999.

[4] Paul Barford, Azer Bestavros, Adam Bradley, and Mark Crovella. Changes in Web client access patterns: Characteristics and caching implications. *World Wide Web*, 1999.

[5] Paul Barford and Mark E. Crovella. Generating representative Web workloads for network and server performance evaluation. In *Proceedings of SIGMETRICS '98*, pages 151–160, July 1998.

[6] Michael Bender, Soumen Chakrabarti, and S. Muthukrishnan. Flow and stretch metrics for scheduling continuous job streams. In *Proceedings of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms*, 1998.

[7] Richard W. Conway, William L. Maxwell, and Louis W. Miller. *Theory of Scheduling.* Addison-Wesley Publishing Company, 1967.

[8] Mark E. Crovella and Azer Bestavros. Self-similarity in World Wide Web traffic: Evidence and possible causes. *IEEE/ACM Transactions on Networking*, 5(6):835–846, December 1997.

[9] Mark E. Crovella, Murad S. Taqqu, and Azer Bestavros. Heavy-tailed probability distributions in the World Wide Web. In *A Practical Guide To Heavy Tails*, pages 3–26. Chapman & Hall, New York, 1998.

[10] Peter Druschel and Gaurav Banga. Lazy receiver processing (LRP): A network subsystem architecture for server systems. In *Proceedings of OSDI '96*, October 1996.

[11] The Apache Group. Apache web server. `http://www.apache.org`.

[12] M. Harchol-Balter, M. E. Crovella, and S. Park. The case for SRPT scheduling in Web servers. Technical Report MIT-LCS-TR-767, MIT Lab for Computer Science, October 1998.

[13] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Héctor M. Briceño, Russell Hunt, David Mazières, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth MacKenzie. Application performance and flexibility on exokernel systems. In *Proceedings of the 16th Symposium on Operating Systems Principles*, October 1997.

[14] D. Karger, C. Stein, and J. Wein. Scheduling algorithms. In *CRC Handbook of Computer Science.* 1997.

[15] S. Manley and M. Seltzer. Web facts and fantasy. In *Proceedings of the 1997 USITS*, 1997.

[16] Vivek S. Pai, Peter Druschel, and W. Zwaenepoel. Flash: An efficient and portable web server. In *Proceedings of USENIX 1999*, June 1999.

[17] M. Pinedo. *On-line algorithms, Lecture Notes in Computer Science.* Prentice Hall, 1995.

[18] Linus E. Schrage and Louis W. Miller. The queue M/G/1 with the shortest remaining processing time discipline. *Operations Research*, 14:670–684, 1966.

THE USENIX ASSOCIATION

Since 1975, the USENIX Association has brought together the community of developers, programmers, system administrators, and architects working on the cutting edge of the computing world. USENIX conferences have become the essential meeting grounds for the presentation and discussion of the most advanced information on new developments in all aspects of advanced computing systems. USENIX and its members are dedicated to:

- problem-solving with a practical bias
- fostering innovation and research that works
- communicating rapidly the results of both research and innovation
- providing a neutral forum for the exercise of critical thought and the airing of technical issues

SAGE, the System Administrators Guild

The System Administrators Guild, a Special Technical Group within the USENIX Association, is dedicated to the recognition and advancement of system administration as a profession. To join SAGE, you must be a member of USENIX.

Member Benefits:

- Free subscription to *;login:*, the Association's magazine, published eight–ten times a year, featuring technical articles, system administration tips and techniques, practical columns on Perl, Java, Tcl/Tk, and C++, book and software reviews, summaries of sessions at USENIX conferences, and Snitch Reports from the USENIX representative and others on various ANSI, IEEE, and ISO standards efforts.
- Access to papers from the USENIX Conferences and Symposia, starting with 1993, via the USENIX Online Library on the World Wide Web.
- Discounts on registration fees for the annual, multi-topic technical conference, the System Administration Conference (LISA), and the various single-topic symposia addressing topics such as object-oriented technologies, security, operating systems, electronic commerce, and NT – as many as twelve technical meetings every year.
- Discounts on the purchase of proceedings and CD-ROMs from USENIX conferences and symposia and other technical publications.
- The right to vote on matters affecting the Association, its bylaws, and election of its directors and officers.
- Discount on BSDI, Inc. products.
- Discount on all publications and software from Prime Time Freeware.
- Savings (10-20%) on selected titles from Academic Press, Morgan Kaufmann, New Riders/Cisco Press/MTP, O'Reilly & Associates, OnWord Press, The Open Group, Sage Science Press, and Wiley Computer Publishing.
- Special subscription rates for Cutter Consortium newsletters, *The Linux Journal, The Perl Journal, IEEE Concurrency, Server/Workstation Expert*, *Sys Admin Magazine*, and all Sage Science Press journals.

Supporting Members of the USENIX Association:

C/C++ Users Journal
Cirrus Technologies
Cisco Systems, Inc.
CyberSource Corporation
Deer Run Associates
Greenberg News Networks/MedCast Networks
Hewlett-Packard India Software Operations
Internet Security Systems, Inc.
JSB Software Technologies
Lucent Technologies
Macmillan Computer Publishing, USA
Microsoft Research
MKS, Inc.
Motorola Australia Software Centre
NeoSoft, Inc.
New Riders Press
Nimrod AS
O'Reilly & Associates Inc.
Performance Computing
Questra Consulting
Sendmail, Inc.
Server/Workstation Expert
TeamQuest Corporation
UUNET Technologies, Inc.
Web Publishing, Inc.
Windows NT Systems Magazine
WITSEC, Inc.

Sage Supporting Members:

Atlantic Systems Group
Collective Technologies
D. E. Shaw & Co.
Deer Run Associates
Electric Lightwave, Inc.
ESM Services, Inc.
GNAC, Inc.
Macmillan Computer Publishing, USA
Mentor Graphics Corp.
Microsoft Research
MindSource Software Engineers
Motorola Australia Software Centre
New Riders Press
O'Reilly & Associates Inc.
Remedy Corporation
RIPE NCC
SysAdmin Magazine
Taos Mountain
TransQuest Technologies, Inc.
Unix Guru Universe

For further information about membership, conferences or publications, contact:
USENIX Association, 2560 Ninth Street, Suite 215, Berkeley, CA 94710 USA.
Phone: 510-528-8649. Fax: 510-548-5738.
Email: *office@usenix.org*.
URL: *http://www.usenix.org*.

ISBN 1-880446-26-X